USENIX Association

Summer Conference Proceedings

Portland 1985

June 11 — 14, 1985
Portland, Oregon USA

USENIX Association

Summer Conference Proceedings

Portland 1985

June 11 — 14, 1985
Portland, Oregon USA

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 U.S.A.
510/528-8649

ACKNOWLEDGEMENTS

Sponsored by:	USENIX Association 2560 Ninth Street, Suite 215 Berkeley, CA 94710 U.S.A.	
Conference Chairperson:	Steve Glaser	Tektronix, Inc.
Program Chairperson:	Greg Chesson	Silicon Graphics, Inc.
Program Committee:	Eric Allman	Britton-Lee, Inc.
	Steve Bourne	Digital Equipment Corporation
	Sally Browning	AT&T Bell Laboratories
	Clement Cole	Masscomp
	Thomas Ferrin	University of California, San Francisco
	Steve Johnson	AT&T Bell Laboratories
	Kirk McKusick	University of California, Berkeley
	Dennis Ritchie	AT&T Bell Laboratories
Tutorial Coordinator:	Michael Tilson	Human Computing Resources Co.
USENIX Conference Coordinator:	John Donnelly	
USENIX Meeting Planner:	Judith DesHarnais	
Vendor Exhibition Manager:	Peter Johnson	Industrial Presentations West, Inc.
Exhibition Technical Advisor:	Randall Frank	University of Utah
Graphics Production:	Ken Serack	
	Michelle Keating	Tektronix, Inc.
	Chris Henick	
	Joe Miller	

TABLE OF CONTENTS

Wednesday, June 12, 1985

PLENARY SESSION

Wednesday (8:30—10:00) **Chair: Steve Glaser (Civic Auditorium)**

Opening Remarks
Conference Organizers and USENIX board

KEYNOTE ADDRESS: **UNIX EVOLUTION:Past, Present, Future**
V. A. Vyssotsky, AT&T Bell Laboratories

Victor A. Vyssotsky is Executive Director of the Research Information Sciences Division of AT&T Bell Laboratories.

Since joining Bell Laboratories in 1956, his work has been primarily in the field of computing. In the late 1950's he was involved in the development of compilers, operating systems and other general purpose software for use with commercially manufactured computers. More recently, he was associated with development of computing hardware and software for the SAFEGUARD ballistic missle defense system. From 1972 until 1980 he worked on computer systems which automate various business procedures of Bell System telephone companies. In his present assignment he is responsible for research in computing science, mathematics, statistics, behavioral sciences and acoustics.

ELECTRONIC DARWINISM

Wednesday (10:30—11:45) **Chair: Greg Chesson (Civic Auditorium)**

LANGUAGES I

Wednesday (1:15—2:45) **Chair: Steve Johnson (Civic Auditorium)**

NETWORKING I

Wednesday (1:15—2:45) **Chair: Mike Karels (Marriott)**

LANGUAGES II

Wednesday (3:15—4:45) **Chair: Steve Johnson (Civic Auditorium)**

NETWORKING II

Wednesday (3:15—4:45) — **Chair: Mike Karels (Marriott)**

TOOLS I

Thursday (8:30—9:45) — **Chair: Dennis Ritchie (Civic Auditorium)**

ARRAY PROCESSING

Thursday (8:30—9:45) — **Chair: Nancy Blachman (Marriott)**

TOOLS II

Thursday (10:15—11:45) **Chair: Dennis Ritchie (Civic Auditorium)**

MULTI-PROCESSING

Thursday (10:15—11:45) **Chair: Steve Emmerich (Marriott)**

OPERATING SYSTEMS I

Thursday (1:15—2:45) **Chair: Kirk McKusick (Civic Auditorium)**

INTERFACES

Thursday (1:15—2:45) **Chair: Sally Browning (Marriott)**

OPERATING SYSTEMS II

Thursday (3:15—4:45) **Chair: Tom Ferrin (Civic Auditorium)**

WINDOWS

Thursday (3:15—4:45) **Chair: Rob Pike (Marriott)**

PERFORMANCE I

Friday (9:00—10:30) **Chair: Clem Cole (Civic Auditorium)**

HARD WORK I

Friday (9:00—10:30) **Chair: Eric Allman (Marriott)**

PERFORMANCE II

Friday (11:00—12:30) **Chair: Clem Cole (Civic Auditorium)**

MAIL

Friday (11:00—12:30) — **Chair: Eric Allman (Marriott)**

HARD WORK II

Friday (2:00—3:30) — **Chair: Greg Chesson (Civic Auditorium)**

COMPILER OPTIMIZATION

Friday (2:00—3:30) — **Chair: To Be Announced (Marriott)**

AUTHOR INDEX

Preface to the Portland Conference Proceedings

This conference marks the Tenth Anniversary of Unix User's meetings. What began as informal gatherings among a handful of people with a common interest has grown into a biannual event of major proportions. Gone are the days when individuals would raise their hands at the beginning of the meeting to indicate they had something of interest to share with the other attendees; our group is just too large for that to be practical anymore. Informal presentations have been replaced by a formal review process complete with conference proceedings available to attendees when they register at the conference. Our meetings of less than 100 people have grown to fifteen to twenty times that size, requiring careful organization and coordination among many people, with plans often beginning more than two years before the actual conference date.

Clearly we have come a long way since those early days. If you're wondering where and when all those past conferences were held and who helped make them happen, here's a list:

When	Where	Host (Chairperson)	Size
10-14 Jun 85	Portland, OR	Tektronix (Steve Glaser)	?????
23-25 Jan 85	Dallas, TX	(Charisse Castagnoli & Rob Kolstad)	1,200
13-15 Jun 84	Salt Lake City, UT	Univ of Utah (Randy Frank)	1,500
16-20 Jan 84	Washington, DC	* (Reidar Bornholdt)	8,000
13-15 Jul 83	Toronto, Can	HCR (Mike Tilson)	1,500
26-28 Jan 83	San Diego, CA	Univ of Calif at San Diego* (Tom Uter)	1,850
6-9 Jul 82	Boston, MA	BBN (Alan Nemeth)	1,300
27-29 Jan 82	Santa Monica, CA	ISC (Mike O'Brien)	1,000
24-26 Jun 81	Austin, TX	Univ of Texas (Wally Wedel)	500
21-23 Jan 81	San Francisco, CA	Univ of Calif at San Fran (Tom Ferrin)	1,000
17-20 Jun 80	Newark, DE	Univ of Del (Dan Grim)	
29 Jan-1 Feb 80	Boulder, CO	Nat'l Ctr Atm Res (John Donnelly)	450
30 Nov 79	Menlo Park, CA	SRI (John Bass)	
20-23 Jun 79	Toronto, Can	Univ of Toronto (Ron Baecker)	400
25-27 Jan 79	Santa Monica, CA	ISC (Steve Holmgren)	350
2 Oct 78	Menlo Park, CA	SRI (John Bass)	75
24-28 May 78	New York	Columbia University (Lou Katz)	~350
12-13 Sep 77	Menlo Park, CA	SRI (Oliver Whitby & John Bass)	~100
19-21 May 77	Urbana, IL	Univ of Illinois (Steve Holmgren)	~250
1-3 Oct 76	Cambridge, MA	Harvard University (Lewis Law)	~120
1-2 Apr 76	Cambridge, MA	Harvard University (Lewis Law)	~60
27-28 Feb 76	Berkeley, CA	Univ of Calif at Berkeley (Bob Fabry)	
31 Oct 75	Monterey, CA	Naval Post Grad Sch (Belton Allen)	
27 Oct 75	New York, NY	City Univ of NY (Mel Ferentz)	
18 Jun 75	New York, NY	City Univ of NY (Mel Ferentz)	40

* indicates joint conference with /usr/group.

Computer Structures are Changing: Will UNIX Change with Them?

C. Gordon Bell
Steve Emmerich
Ivor Durham
Daniel P. Siewiorek
Andrew Wilson

Encore Computer Corporation
15 Walnut St.
Wellesley, MA 02181
uucp address: encore!emmerich

ABSTRACT

The UNIX operating system does not adequately support a significant, emerging computer structure: the Multi (symmetric shared memory, multiple microprocessor) computer. This paper offers reasons why the Multi, in particular, is a significant computer structure, and describes some of the reasons that developing and executing applications built to use multiple cooperating processors will require facilities beyond those found in UNIX. We argue (a) that multiprocessor and distributed computers for which UNIX is not currently well suited will dominate in the next decade; (b) that in order to make UNIX and the notion of a popular, non-proprietary OS survive, the UNIX technical community should focus on providing appropriate support for these new computing structures; (c) that progress towards such support should be shared in a forum sponsored by USENIX (called the "Multi- and Distributed UNIX (MAD- UNIX) forum"), so that progress is in the hands of a coalition of sponsors, not in the hands of a single company, in order to avoid the tendency for the standard to become proprietary or to ossify.

1. The Multi -- An Important New Architecture

One important, emerging computer structure is the Multi, or multiple microprocessor computer.

Multiprocessors consist of two or more processors capable of independent instruction execution and able to access programs and memory held in a common, shared memory. Mainframes have been built in multiprocessor configurations with two to four processors, such as the Burroughs B5000 (a dual symmetric multiprocessor), the IBM System 370 and 308X series computers (ranging up to a quad processor). Larger multiprocessors were not successful (a) because switching and cabling cost overhead grew as the product of the number of processors and memories, and (b) because due to program sharing, a multiprocessor with N processors requires faster, more expensive memory than a uniprocessor requires, whose cost exceeds that of N separate, slower memories.

The Multi composed of microprocessors alleviates the cabling/switching cost issue by using a fast, inexpensive and short central switch or "bus" for all communication between processors, memories, and input/output devices (impossible for physically larger mainframes). Advances in

memory cacheing technology addresses the memory cost issue by providing a fast memory subsystem using high speed, expensive cache memory and lower speed, cheap main memory, while in many implementations allowing for full data concurrency between all caches in a system.

As a result, the small size, low cost and rapidly increasing performance of microprocessors enable the design and construction of computer structures that offer significant advantages in the design, manufacturing price-performance ratio and reliability over traditional computer families implemented from TTL and ECL semiconductor technologies. Current, commercial Multis typically consist of 2-28 microprocessors, common memories and input/output devices which communicate across a fully shared bus structure. [Multiprocessors with different interconnection schemes between components also exist, but they tend to be specialized for high performance in limited application areas.]

There are several commercial Multi's, built by companies such as Sequent, Arete, Synapse, Intel, and Encore, each with different processor-memory-in-out interconnect, configuration capacity, and cost. As an example, Encore's Multimax is a Multi designed for high- performance processing and high input-output data rates. Up to 20 processors with caches can be plugged into the bus (on ten modules), along with up to 32 megabytes of completely shared main memory. The bus and system architecture permits 1024 processors to address 4 gigabytes of physical and virtual memory, permitting expansion as higher degrees of component integration are achieved. The system's bus, the Nanobus, transmits 100 megabytes per second, sufficient for anticipated increases in speed of CPU chips (and resulting increases in bus loading from memory accesses) for many years. Terminal and workstation access is via one or more local-area networks.

2. Operating Systems on Multi's

The Multi structure is well-suited to high-speed of data connectivity between multiple CPU's (via shared memory), and fast synchronization between processors (via test-and-set type operations, and interrupts). An operating system can achieve high throughput in real-time, timesharing and transaction processing applications, by dynamically distributing jobs among CPU's. In a multiprogramming (timesharing) environment, throughput (the number of jobs completed per unit time) is incrased by dividing processes among the many processors in a way that is transparent to users. In real-time applications, one or more CPU's can be dedicated to device polling and/or interrupt handling as well as data transfer to mass storage, while other processors control the general programming and administrative environment. Since each process will typically spend 25 to 50 percent of its time in the UNIX kernel, UNIX must be adapted to allow multiple processors to be operating in the kernel simultaneously (as AT&T, Sequent, and Masscomp have described at these conferences). For many scientific and engineering applications in which data sets can be partitioned and operated upon in parallel, multiple CPU's can be applied to a single problem simultaneously to achieve speedup, enabling the Multi to get results at the speed of much more expensive computers, but at a fraction of the cost. Likewise, in commercial transaction-processing and database operations, processors can be working in parallel through partitioning of datasets or transaction steps to increase transaction rates and improving response time.

3. Applying Multiple Processors to Speed Up a Single Job

Consider the following idealized scenario on the development of a parallel program. There is a critical application program on a uniprocessor which has become a productivity bottleneck (e.g. VLSI design rule checker, partial differential equation solver, etc.). Dynamic parameters from the uniprocessor version are measured, such as instructions between accesses to key global data. The parameters are utilized in existing performance prediction models to determine the number of processors that can be effectively utilized. With the optimum number of processors as a guideline, the user creates a suitable parallel decomposition for the algorithm, and sets up a

simulation with automatic generation of a synthetic workload. The instrumentation, data collection, and data analysis primitives and tools are used to locate bottlenecks and contentions. Once the synthetic workload has been balanced, the programmer replaces each synthetic process by actual code.

Clearly, the desire to exploit concurrency within a particular piece of software increases the complexity of the overall task. In the experience of researchers at CMU, there are two major differences between sequential and parallel programming. The first is the requirement that the speed of data communication and synchronizaton between processors take advantage of the speed of the shared memory between processors, to maximize aggregate performance. The information communicated between processors is comprised of raw data to be processed, plus information to synchronize tasks so that the integrity of the information is maintained. The second difference is the control of concurrent activities. Controlling concurrent activities involves not only scheduling tasks so that the "producers" deliver information in time to keep the "consumers" busy, but also the need to stop and start collections of activities together, and to avert undesirable interactions including both contention and deadlock in the sharing of resources. Both communication and control issues have been addressed in distributed systems interconnected by Local Area Networks. However, the high-overhead solutions for loosely coupoled LAN's are not appropriate for tightly coupled shared memory multiprocessors. Communication and control on multiprocessors has been the subject of study on C.mmp and Cm* at CMU, Ultracomputer at NYU, and TRAC at Texas. Experience shows us that special scheduling algorithms, IPC mechanisms, data abstractions, and tools are needed to assist with parallel program development and execution. (A bibliography on these topics are being distributed at the conference, separate from the Proceedings).

The complexity of developing parallel programs suggests that a "Parallel Programming Environment" needs to be devised, in which each step of design, specification, implementation, instrumentation and debugging are supported by tools.

The design stage should provide various models of computation including proven templates for typical techniques of parallel processing. Some of these are as follows: parallel decomposition (e.g. master-slave, pipeline, asynchronous, synchronous, multiphase, partitioning, and transactions); templates for high availability and fault tolerance (e.g. journaling, shadow processes, duplicate and compare, replicate and majority vote); and performance prediction models (e.g. speed-up prediction as a function of parameters measured on a uniprocessor version of an algorithm). Because performance will be even more sensitive to design in a parallel environment, tools for creating synthetic workloads and simulating multiprocessor contention (both hardware and OS) would be useful.

Tools for defining data abstractions and their functionality, and the synchronization mechanisms for those abstractions, would assist in the specification, implementation, and debugging phases. Libraries of such abstractions could be built up that have well understood cost, performance, and reliability characteristics. Extensions to programming languages should be provided to describe concurrency. Structure editors would be available to assist writing of syntactically correct code, and compiler and run-time checks should be included for semantic checking. The debugging and instrumentation stage should be supported by facilities for managing multiple activities, e.g. starting, stopping, and scheduling of events, as well as notification upon detection of a set of events.

4. How can we Cooperate To Support Distributed and Parallel Processing?

Several vendors have adapted UNIX to multiple processors by allowing multiple threads of execution through the kernel, but this level of support does not address how the aggregate computing cycles of Multi's can speed up the time to perform single applications that are organized as multiple tasks or processes. Research in alternative computer architectures, operating systems, languages, and algorithms is not likely to provide answers in the near future. This is up to us, the UNIX community, to learn and adapt to UNIX technology.

Until now, the motivation for changing UNIX has properly been to make it more fully realize its original intended purpose of being a better, more portable timesharing environment that exploits its underlying hardware, particularly for software development and technical documentation preperation applications. Support for evolutionary hardware developments such as virtual memory and peer-to-peer networking were successfully added, largely as a result of public domain, government-funded efforts at U.C. Berkeley. More recently UNIX has been adopted as a workstation development and execution environment, and as a multi-tasking OS for PC's.

It is ironic that after having been a source of much creative change in OS development over the past 15 years, UNIX(tm) is commercially supported and subject to pressure to be standardized when it is least suitable for supporting the fastest-growing computing styles and most cost-effective hardware structures. Networking, bitmap displays, and multi-processors are the new components of these computing structures and styles which UNIX does not succeed in managing very well. In order to manage these hardware resources properly, UNIX clearly must support integral networking (or better capabilities for integrating network- based systems capabilities into the operating environment), and flexible distributed resource sharing and control between heterogeneous machines. Progress is being made in this direction, notably by SUN and AT&T and numerous universities. More progress could be made towards supporting closer control by database and transaction processing systems over process and task scheduling, and disk access, for improved performance, as these are areas where much research and practical systems building has taken place since the UNIX filesystem was invented. In addition, UNIX must provide suitable constructs for supporting diverse bitmap displays that provide heterogeneous graphics services and windowing facilities. This is an area in which must experimentation is occurring, as it is not very well understood yet how screen objects, operating systems, and languages ought to interact for good performance and flexibility.

Interested members of the USENIX Association could make a difference in addressing distributed and parallel processing needs in UNIX, and in helping to shape a good development and execution environment for distributed and multiprocessors with UNIX, by joining Encore in the founding of the "MAD-UNIX forum." We would like to discuss with you appropriate mechanisms for participants to share ideas, over the network or in person.

It is our belief that if UNIX does not evolve fast enough to support distributed resource sharing across networks, and parallel applications on Multi's, other proprietary OS's will. Given how significant we believe the Multi structure to be, this could mean that UNIX could become AT&T's operating system for their computers, and merely a "compatibility interface" for other successful computers. Though possibly inevitable, the loss of momentum behind UNIX as an OS for the future (rather than the past) would be a loss for all of us who believe that UNIX can continue to be a source of creative, practical ideas in computing.

UNIX* on a Microprocessor - 10 Years Later

Heinz Lycklama

INTERACTIVE Systems Corporation
2401 Colorado Avenue, Third Floor
Santa Monica, CA 90404

213-453-UNIX
decvax!cca!ima!heinz

ABSTRACT

Ten years ago a subset of the UNIX system was first ported to a microprocessor-based system (LSX)**. Today the UNIX system is readily available on a variety of microprocessor-based systems. In the last decade, there has been a tremendous technological advance in the various hardware components which make up a system suitable for supporting UNIX on a personal computer. The changes in the UNIX system software over the last 10 years have impacted the implementation of the UNIX system on personal computers and vice versa. We take a look at seven different implementations of UNIX on a microprocessor over the last decade and then examine the impact that new technology advances are likely to have on the personal UNIX system in the coming years.

1. Introduction

A subset of the UNIX system was first ported to the LSX microprocessor-based system 10 years ago, thus becoming the first single user UNIX system. Today the UNIX system is readily available on a variety of microprocessor-based systems. Four of the more popular microprocessors on which the UNIX system is implemented are the Intel 8086, Motorola 68000, National 32000, and the Zilog Z8000. Including minicomputers and mainframes, the UNIX system has been implemented on many proprietary machine architectures as well. The portability of the UNIX system has made it relatively easy to port the complete system to new processors as they are introduced to take advantage of the new technology available today. This has made the transition from timesharing minicomputers to single-user personal computers possible in a relatively short period of time.

There are really three distinct markets for personal UNIX systems: the workstation, home computer system, and office computer system. The workstations are at the high end of the market where applications such as CAD/CAM and VLSI design require powerful single-user workstations with sophisticated graphics capabilities for engineers and designers.

---

* UNIX is a trademark of AT&T Bell Laboratories.

** UNIX on a Microprocessor, H. Lycklama. The Bell System Technical Journal, Vol. 57, No. 6, July-August 1978.

The personal UNIX system in the home environment will be used by professionals requiring sophisticated packages, while the system in the office will be used by professional software developers as well as the less-sophisticated office worker. We restrict our comments in this paper to the trends in the personal UNIX system for the home and the office.

When the LSI-11 was introduced by DEC in 1974, it was one of the very first microcomputers made available by a computer vendor. This was at a time when the UNIX system had not yet been ported to any system other than various models of the PDP-11 computer, each with a memory management unit and with I/0 peripherals supported on the 16-bit wide UNIBUS. The LSI-11 microcomputer did not have a memory management unit and supported I/O peripherals on the 16-bit wide QBUS with multiplexed address and data lines. The instruction set of the LSI-11 micro is a subset of that of the PDP-11. The memory available and the secondary storage available provided a real challenge in porting enough of the UNIX system to this hardware to make a useful system. The LSX system was diskette-based and had a maximum of 56K bytes of memory available to run in. 16K bytes were used by the operating system itself, making a maximum of 40K bytes available to user programs.

Ten years after LSX was implemented, HP announced the availability of its Integral Personal Computer, using an 8MHz 68000 microprocessor, with 768K bytes of main memory, 710K bytes of diskette storage, and its UNIX System III based operating system, HP-UX, burned into ROM. This is one of the first portable UNIX systems. From LSX to HP-UX on the portable computer, there was a factor of 10 increase in the size of main memory available but not a correspondingly large change in the amount of secondary storage available. The HP-UX system does use a memory management unit, however, making true multi-tasking possible. There was roughly a five-fold increase in instruction speed.

There have been some significant changes in the technology available on minicomputer-based systems since the LSX system was first built and demonstrated in June 1975. The evolution of hardware technologies and of the UNIX system itself have greatly impacted the power available in a personal UNIX system. Such UNIX-based systems as ONIX, FOS, Xenix, PC/IX, HP-UX, and the recent UNIX PC from AT&T, are discussed in some detail.

The software provided on microcomputers has followed the release of different versions of the UNIX system from AT&T. However, vendors have provided various enhancements to the system to provide systems suitable for the commercial environment. We discuss features such as IPC (semaphores, shared memory and messages), MMU support (protection, relocation, and paging), contiguous files, real-time support, Xenix enhancements for micros, PC/IX enhancements for micros, communications, and record/file locking.

The evolution of technology has made the implementation of a personal UNIX system practical and inexpensive during the past few years. We discuss some new technologies and how they are likely to impact the wide availability of personal UNIX systems in the next few years.

2. Hardware Technology

The minimum hardware components required to support a personal UNIX system today are the CPU, memory, diskette storage, fixed disk storage, a display terminal, and possibly some communication hardware, such as an asynchronous RS232 line and a modem. Each of these components has undergone a tremendous change in the last decade and has contributed to the ready availability of an inexpensive personal UNIX system.

2.1 Microprocessor Technology

The LSI-11 microprocessor was the first of its kind at the time it was introduced by DEC in 1974. Its instruction set was compatible with that of the PDP-11, the first processor for which UNIX was implemented in the C language. The underlying I/O structure for the LSI-11 was only slightly different from that of the PDP-11. The LSI-11 was a 16-bit processor and had a MIPS (million instructions per second) rate of less than 0.2. The LSI-11 did not have a memory

management unit, making relocation and protection impossible. It also did not support split instruction and data space, thus restricting the size of a user program to a total of 56K bytes since the upper 8K bytes were dedicated to I/O device addresses.

One of the next major microprocessors introduced was the Z8000 by Zilog in 1978, another 16-bit micro, and an evolutionary step up from their 8-bit Z80 micro. The Z8002 did support split instruction and data space, allowing larger programs to be supported. It is very well suited to UNIX and C, leading to an early version of the UNIX system being ported to this architecture. Zilog also provided the Z8010 memory management unit but early microcomputer systems used specially-built memory management units. The Z8002 microprocessor had a MIPS rate of about 0.5.

The first 16-bit microprocessor introduced by Intel was the 8086, upwards compatible with the 8080 instruction set. Support for segmented memory is available, but an additional MMU is required to provide protection in a multitasking system such as UNIX. Twenty-bit addressing is available on the 8086. It and other members of that family, such as the 8088, are the predominant microprocessor chip used in the IBM PC and PC compatibles. The MIPS rate of the 8088 is not very different from the LSI-11, while that of the 8086 is up to three times greater.

The other significant microprocessor used by UNIX systems is the Motorola 68000, a 32-bit micro with a 16-bit data path. It has become known as the UNIX micro because of its dominant position and the fact that it is well suited for the UNIX system. The current version of this microprocessor, the 68010, supports virtual memory.

There are other microprocessors available or on the drawing boards that are suitable for supporting the UNIX system, but they are not yet popular in the personal UNIX marketplace. They include the National Semiconductor family of micros, the 16016, 16032, 32032. These micros are well suited for UNIX, but the selection of support chips has been insufficient to lead to an inexpensive UNIX machine. The next generation of Zilog, Intel, and Motorola chips: Z80000, 286 and 386, and the 68020, respectively, will also be available in the next year or two and will offer MIPS rates in excess of 1.0, about three times that of the 8086.

2.2 Memory Technology

At the time that the LSI-11 was introduced, core memory was still quite popular. However, microprocessor-based systems were starting to use MOS memory. The LSI-11 system described here used MOS memory with 4Kb chips. In the early 1980s, 64Kb chips were prevalent and today, in 1985, commercial products are using 256Kb chips. This represents a 64-fold increase in density of memory cells over the last decade. A 512 kilobyte memory board can be purchased for a microcomputer for under $600 today. In 1975, $1000 could buy about 8K bytes. This represents about a 128-fold increase in value. As memory densities increased, the memory access times were reduced. From 1975 to 1985, there was roughly a fourfold reduction in memory access time from 800 nsec. to about 200 nsec. today.

2.3 Diskette Technology

Diskettes were manufactured in the 8" size in 1975. The capacity of the diskettes used with the LSX system was about 250K bytes. In 1981 when the IBM PC was introduced, the popular diskette size was 5-1/4" with a capacity of 360Kb. In 1984 with the introduction of the PC AT, the 1.2Mb 5-1/4" diskette was used. The 3-1/2" diskettes are now also starting to appear on the market. Overall, the diskette capacity has quadrupled and the cost has been cut by at least a factor of two. This has resulted in cost per byte reduction from $4 per kilobyte in 1975 to 50 cents per kilobyte in 1985.

2.4 Disk Technology

At the time of the introduction of the LSI-11, Winchester disk technology was not very prevalent for minicomputer and microcomputer systems. When the ONYX system was introduced in 1980, it came with a 10 megabyte fixed 8" disk, barely enough to hold a complete UNIX binary file system. The IBM PC XT was introduced in 1983 with a 10 Mbyte 5-1/4" hard disk, whereas the IBM PC AT was introduced in 1984 with a 20 Mbyte 5-1/4" fixed disk. Larger Winchester disks are available today, but are not typically provided with the small microcomputer systems. As with diskettes, the fixed disks are also being made in the 3-1/2" size today. In general, the recording density of fixed disks has gone up by a factor of 10 every eight years. This was accompanied by a cost reduction from 80 cents per Kb in 1980 to about 10 cents per Kb in 1985.

2.5 Display/Terminal Technology

The number of terminals available in 1975 was much less than today, with the average price being somewhat over $1500 for an ASCII terminal that could display 24 lines of 80 characters. Today prices for such terminals have come down by at least a factor of three. In addition, the alphanumeric terminals available today are programmable and have enough RAM memory to hold a few pages of text. In 1975, terminals were not typically programmable. Connection to the computer was almost always by means of an RS232 interface at 4800 to 9600 baud. Today such terminals can be connected to the computer at 9600 to 19,200 baud. Now it is common for the personal computer to have a bitmap display terminal directly connected to the computer on the memory bus so that display of text and data is "instantaneous." With the advent of inexpensive RAM and graphics controllers, display terminals are prevalent in personal computers. Today almost all personal computers have one memory-mapped display terminal with a few asynchronous terminal lines available for logging in and exchanging data with other computers.

2.6 Communications Hardware

In 1975, the common communication line was a RS232 interface allowing computers to interchange data. Today these interfaces are still prevalent, but high speed communication busses are becoming more popular for large data transfers. The Ethernet interface has been available for the IBM PC for a few years now, but it is rather expensive to use and the 10 Mbit/sec data path is an overkill for personal computers. The PCNET hardware supports 2 Mbit/sec data bandwidth using the CSMA/CD broadband technology. More recently, AT&T has announced the Starlan network which uses the CSMA/CD baseband technology also used in Ethernet, but running at 1 Mbit/second, using existing telephone wiring in many locations. The data rates supported by these latter two communication products are sufficient for most personal UNIX systems.

3. Personal UNIX Systems

Some of the more popular personal UNIX systems are described to give the reader a sense of the trend of these systems in the last decade. In 1975, the Version 6 system of UNIX was available. In 1985, System V Release 2 Version 2 has just been released. The following systems will be discussed in more detail to give the reader a historical perspective.

System	Processor	UNIX Base	Date
LSX	LSI-11	V6	June 1975
ONIX	Z8000	V7	1980
FOS	68000	V7	1982
Xenix	8088	V7	1983
PC/IX	8088	Sys III	Jan. 1984
HP-UX	68000	Sys III	Dec. 1984
UNIX PC	68010	Sys V	Mar. 1985

3.1 LSX System

The LSX system was developed out of a need to provide a self-contained portable system for supporting a sophisticated music synthesis system. At this time the LSI-11 microprocessor was introduced by DEC on a single board. However, all of the components required to build a complete computing system were not conveniently packaged and available off the shelf. Thus a homebrew computer system was designed and built to support a standalone UNIX system. The basic system consisted of a LSI-11 microcomputer with 40K bytes of memory, a diskette controller with two drives, a serial interface card, and the associated power supply and cabinet.

The lack of a memory management unit meant that executables had to be relocated to run in memory, starting at 16K, since the operating system itself occupied 16K bytes of memory, including the system buffers. User programs were constrained to run in 24K bytes. Even the C compiler ran in that amount of memory. The only major program which had some size restriction was the yacc compiler-compiler. It was possible to recompile the complete system kernel sources on the LSX system itself.

To fit the UNIX kernel within 16K bytes of memory, some functionality had to be given up. Pipes were not implemented in the kernel, but were simulated at the user level by using intermediate temporary files, with a simple change to the shell command interpreter. Profiling and interactive debugging of processes were not supported. Complete file protection was not provided, and only one character and one block file type was supported. Mounted file systems were supported, but not dynamically. Given the slow diskette data transfer rate, processes were run to completion before swapping in the next runnable process. The number of processes that could be supported was limited to a total of three, with the allowance of one background process.

Some unique features were added to the LSX system to enhance performance and to tailor the system to the restricted hardware environment. The standard V6 UNIX file system was supported, but in addition, contiguous files were supported by means of a new file type. The contiguous files were represented by extents in the inode. The mixture of these file types was made possible by the fact that blocks were allocated to files from a map, rather than from a linked list. The layout of the data structures on disk enabled the optimization of disk performance. Loading of an executable file was dramatically improved over that of an unmodified file system.

The size of the diskettes limited the number of files which could be stored on any one diskette. This led to the requirement to subset the system. Typically one diskette contained a software development system with the C compiler and associated tools and another contained text processing tools. For a two diskette system, the user would load either the software development diskette or the text processing diskette on the system disk, and use the second diskette for user files. The system was an attempt to package a personal UNIX system for use by the individual software developer. It was successful in its purpose and has led to other useful innovations.

3.2 ONIX System

The LSX system was never released as a commercial product, although the demand was there, mainly because the license fee for UNIX was $20,000 and there was no such thing as a binary license. It was only with the introduction of V7 UNIX that a binary license as low as $100 was introduced by AT&T. The first commercial UNIX system based on a microprocessor was the ONYX Z8000-based ONIX System based on V7 UNIX in 1980. Although strictly speaking it was not a single-user system, it was the first commercial microprocessor-based UNIX system and was the forerunner of many personal UNIX systems in the next five years.

The ONYX hardware consisted of a Z8002 processor board with a specially-built memory management unit that enabled the protection and relocation of code, 256Kb of RAM using 16K chips, a 10 Megabyte hard disk and a 1/4" cartridge tape unit for backup. Peripherals were controlled by a Z80-based I/O board.

The software on the ONYX system was based on the V7 system released by Bell Laboratories in 1979. The major addition made to the software was the file/record locking primitive to support commercial application packages. Other than that, the software was rather vanilla V7 UNIX, suitable as a software development system and for support of special vertical application packages. Not much off-the-shelf software was available for UNIX systems in 1980. At $20,000, the system was too expensive as just a personal UNIX system, but it did establish a new market for low-cost UNIX systems.

3.3 Fortune FOS

The Fortune system was introduced in 1982 as a low-cost 68000-based UNIX machine. This was really the first attempt to break the $10,000 price barrier for a UNIX-based system. The hardware consisted of a 68000 microprocessor with a memory management unit and a minimum of 128Kb of RAM memory. Secondary storage options consisted of an 800Kb diskette and a 10Mb hard disk. Streamer tape was available as a backup option. A monochrome display was provided as the console device. Additional RS232 lines were available for logging in other users and for communicating with other systems. Additional hardware devices were supported for high-speed communication.

Although the system was based on V7 UNIX, file/record locking support was provided for some commercial application packages. A user-friendly menu interface was provided to hide the more esoteric features of the UNIX interface. The memory management unit provided support for multiple users. Thus, this was more than a personal UNIX system. The concept of unbundling the UNIX system software was also new with the Fortune system. Program development tools were an extra cost option.

3.4 Xenix

The Xenix system, developed by Microsoft, was a generic system developed for a number of microprocessors, including the PDP-11 microcomputer. Xenix was ported to the IBM PC/XT system by Santa Cruz Operations and made available as a low-cost UNIX system. The IBM PC/XT consists of an 8088 CPU board (without a memory management unit), up to 640Kb RAM, 360Kb 5-1/4" diskette, 10Mb hard disk, memory-mapped monitor, and up to two RS232 communication lines. At the time of its introduction, the PC/XT was introduced in its basic hardware configuration for about $5000.

When initially introduced, Xenix was based on V7 UNIX. In 1984 it was updated to be UNIX System III compatible. Xenix is an enhanced version of the UNIX system, with file/record locking, shared memory, semaphores and synchronous "write" support for commercial applications. Automatic file recovery and file system integrity checks have been added. The user interface has been enhanced for the end user and software developers by means of the "visual shell" command. Various popular UC Berkeley enhancements, such as the vi editor and the C shell have been added to Xenix as well. Attempts have been made to tune the performance of the system to the microcomputer environment. For the microcomputer user, the system is unbundled into three subsets: the operating system, the software development system, and the text processing system.

3.5 PC/IX

PC/IX was the first UNIX-based system advertised as a single-user UNIX system for the commercial environment. It was developed by INTERACTIVE Systems Corporation and marketed by IBM, and runs on the basic IBM PC/XT system with 10 megabytes of hard disk. The PC/IX system is based completely on UNIX System III with some INTERACTIVE-supplied enhancements. The popular file/record locking primitive was added to the system for commercial applications. A two-dimensional screen editor with function-key editing and "help" facilities was provided as a user-friendly editor as a standard part of the system. Co-residence with PC-DOS was provided, with the ability to transfer files between the UNIX and the PC-DOS environments, a very important feature, since PC-DOS is the predominant operating system environment on the

PC. A general-purpose queuing system was provided for control of output streams of data.

PC/IX was a true personal UNIX system developed specifically for the IBM PC/XT. It has been highly performance tuned to take advantage of, or circumvent the architecture of the PC machine. The two-dimensional screen editor used direct memory-mapped I/O to achieve superior performance, rather than the terminal-independent display package available with the Berkeley software distribution. Pseudo-contiguous files are used for executable files and for data files where feasible. This enables the system to load executable images directly into the user's address space for efficiency reasons. Read-ahead of data from files was implemented for maximum performance. Partial swapping was implemented to minimize data movement between the disk and computer main memory. Specially-tailored enhancements such as these, enabled PC/IX to outperform all other UNIX implementations on the IBM PC/XT.

3.6 HP Integral Personal Computer

The HP Integral Personal Computer was the first truly portable computer with a UNIX system. It used an 8-MHz Motorola 68000 microcomputer with a minimum of 512Kb of RAM memory, one 3-1/4" 710Kb microfloppy, a special 9" electroluminescent screen and a lightweight printer. The UNIX kernel is actually provided in ROM. This was the first commercial UNIX system offering without a hard disk. In that sense this system is very akin to the LSX system implemented 10 years ago.

The HP-UX system is a superset of UNIX System III. The kernel, the personal applications manager (user application interface), and the window manager require 256K bytes of ROM. With the limited secondary storage, software bundled on disk includes a tutorial, system utilities, 31 standard UNIX commands, customer diagnostics, and some standard applications such as editors, games, and fonts. Additional UNIX commands are, of course, available on separate microfloppies. Up to 50 percent of the RAM memory can be allocated as a RAM disk to speed up application program performance. The bitmapped terminal (255 x 512 pixels) is an integral part of the system. The complete system is compact and marks a new milestone in UNIX system availability to the personal computer user.

3.7 AT&T UNIX PC

It is perhaps ironic that one of the latest personal UNIX systems introduced was the AT&T UNIX PC, almost 10 years after the LSX system was developed within Bell Telephone Laboratories, a part of AT&T. This is also the first personal UNIX system based on UNIX System V. It uses the 10 MHz 68010 microprocessor with a minimum of 512K bytes of memory, 320Kb diskette, 10Mb hard disk, and a 720 x 334 pixel monochrome monitor. Custom memory-management hardware provides virtual memory support using 4K byte pages. One of the features which distinguishes this personal UNIX system from all others is that telephone functions are integrated into the system.

In terms of new software features, this is probably the most that has ever been put into a personal UNIX system. The system is based on UNIX System V, revision 2 with demand-paged virtual memory, shared libraries, and a window- and menu-driven software environment that allows the new user to access computer-assisted telephone functions, UNIX functions, and other optional application programs. Use of icons is also possible in the system. A real attempt has been made to provide a user-friendly interface to the new user. Software is divided into two sets - the System Software and the UNIX utilities.

4. Summary of Changes in Last Ten Years

The last 10 years have seen tremendous technological changes in the hardware and software components that make up a personal UNIX system. Each one has impacted the power of the system available to a UNIX user, but some much more than others.

4.1 Changes in Hardware Technology

The speed of the microprocessor chip used for the personal computer has changed by about a factor of five from the LSI-11 to the 68010. The density of memory chips has increased by a factor of about 64. Thus larger memories become more economical. Fortunately, we have migrated from 16-bit microprocessors to 32-bit microprocessors enabling access to larger memories. Whereas the first personal UNIX system required less than 40K bytes for system and user, today 256K bytes is the minimum and one megabyte is not uncommon. That's an increase by a factor of 25, but with a price per byte reduction by a factor of about 128.

Diskette capacity has not increased much over the last decade. LSX used 256K byte diskettes while PC/IX on the PC AT used 1.2M byte diskettes. That's an increase by a factor of four with a factor of eight reduction in cost per kilobyte. Winchester disks became popular for micros in 1980. The size of disks used for personal UNIX systems has not increased much over the last five years, although larger Winchester disks are available. Cost per byte of storage has decreased by a factor of eight over the last 10 years.

In 1975, "dumb" terminals were the norm for small computers. Today monochrome (or even color) bit-mapped terminals are the standard for the personal UNIX system, as the console terminal. This has improved the response time for screen-oriented applications as well as for the standard UNIX programs.

Networking was non-existent at the time LSX was implemented. UUCP was introduced in the late 1970s for communication between UNIX systems over asynchronous lines. Today the 2M bit broadband PCNET and the 1M bit baseband Starlan network have been introduced for the personal UNIX systems. However, they have yet to be well integrated and supported by the software vendors.

4.2 Changes in UNIX Software Technology

When one looks at the changes made in the UNIX-based software, the improvements are not as dramatic or obvious as in the hardware technology. The kernel for V6 UNIX was about 40K bytes in size. For LSX it was 16K bytes. Today the System V kernel is well over 100K bytes in size, even for the personal computer version. That is more than a six-fold increase over the LSX kernel.

The major additions have been in the IPC mechanisms - pipes, messages, semaphores, and shared memory. The terminal driver has certainly grown much more sophisticated and complex. More I/O buffers are provided in the kernel to provide better I/O throughput. The latest version of the UNIX system supports virtual memory, file/record locking, and job control, making the kernel larger still. Networking is not yet a standard part of the UNIX kernels, however.

The V6 UNIX system had about 35 system calls; the latest versions have about 55. There has not been an uncontrolled addition of new system call interfaces to the kernel, but many of the system calls themselves have grown in complexity. The "open" system call, for example, has many more options now than it had in the V6 version of the UNIX system. The ability to trace related processes has been added via a new "ptrace" system call. Accounting capabilities have been added. The "exec" system call has added the environment capability. The interprocess communication (IPC) system calls have added significantly to the complexity and size of the kernel code. It is difficult to argue that the functionality available to the user has increased in proportion to the increase in size of the kernel code. Some of the later additions, such as virtual memory support and file/record locking are not likely required for many personal UNIX applications.

The LSX system, in its basic form allowed one to run 24K byte user programs. At most one could run 40K byte user programs. The 8088 and Z8000 processors allowed one to run programs with up to 64K bytes of code and 64K bytes of data each. The 68000 processor allowed one to run programs as large as the physical main memory minus the size of the kernel itself. With the 68010 and the support of virtual memory on the UNIX PC, one can run user programs which are

a few megabytes in size.

The amount of code, as measured by the number of bytes occupied on secondary storage, has increased by about a factor of three or four. The V6 UNIX file system containing all system and application code occupied about 2.5 to 3 megabytes. Today the complete UNIX system software distribution occupies more that 10 megabytes of secondary storage. The "printf" function, included with most programs, has grown by a substantial amount. The "stdio" library routines have increased in size significantly. The sizes of many of the programs have also increased due to added functionality. And of course the number of programs comprising the UNIX system has more than doubled since the early edition of the UNIX system.

There have been some significant functionality additions in the basic UNIX software. With the V7 UNIX system, 32-bit file systems were introduced, allowing more users per system and access to much larger file systems. Some basic networking commands were also added. These include the unix-to-unix copy commands (uucp) and the virtual terminal connect program. The lexical analysis program (lex) was added, as was the general program configuration facility "make". Text processing facilities were enhanced and some primitive graphics capabilities were added. The "spell" program was also introduced at this time. Although the PDP11 UNIX system did not use it in production, the portable C compiler was introduced as well. This formed the basis for producing a family of C cross compilers targeted for different computers. Eventually all of the "microports" used this as a base. But a price had to be paid for this portability. Whereas the C compiler running under LSX occupied a total of 24K bytes for both code and data, now the portable C compilers require close to 64K bytes just for code. In the System III distribution of the UNIX system, one of the key additions was the Source Code Control System (SCCS) and the Bourne shell. Device-independent troff was made available with the introduction of System V, as was the "vi" screen editor from Berkeley with the associated screen control package of subroutines and terminal data. Shell layers were introduced with the second release of System V.

The UNIX system has always been regarded as an excellent system for program developers. However, it provides a difficult-to-use environment for naive computer users. There has been an attempt with subsequent releases of personal UNIX systems in particular, to improve the ease-of-use for new users. Even with LSX, some improvements were made in the system management facilities so that the new user would not have to remember a complex procedure in order to bring the computer up or take it down. When first booted, the diskettes in the system were automatically mounted. Upon the completion of each command, file inodes were updated on disk so that if the computer was taken down abruptly, the file system on disk would remain intact. The "fsck" program was available with the ONIX system for the first time to help with the repair of broken file systems. File system robustness features were added to later releases of the UNIX system to minimize the possible damage to file systems when an unexpected computer crash occurred. Installation and maintenance of a system has become easier with later releases of the personal UNIX systems. The PC/IX installation procedures are largely menu-driven, and booting and taking down the system invoke file system checking programs automatically.

The UNIX system has had a bad reputation in the commercial marketplace for an unfriendly user interface. Not much was done to improve on this for either LSX or the ONIX systems. However, the Fortune FOS system made substantial improvements in this area. The system was largely menu-driven, and made use of windows in the presentation of information to users. The Xenix system provided a "visual shell" for users. The PC/IX system introduced the TEN/PLUS* user interface in an attempt to provide the same interface for many application programs in an ASCII environment. The UNIX PC from AT&T makes heavy use of windows and a menu-driven interface, using the bitmapped terminal provided as the console. The HP IPC also uses window management software and a menu-driven interface. It is possible to hide the shell command interpreter.

* TEN/PLUS is a trademark of INTERACTIVE Systems Corporation.

With the increase in the size of the UNIX programs and the need to package software in smaller pieces than 10 megabytes, personal UNIX systems have always had to contend with packaging subsets of programs. The LSX system only had 256K bytes on one diskette and therefore had to provide a different system disk for a development system than for a text processing system (for producing documentation). The Xenix system subsetted the software into three pieces: system software, program development system, and a text processing system. The PC/IX software was packaged so that a user did not have to load all of the UNIX software on the 10Mb fixed disk. The UNIX PC software from AT&T is packaged into two pieces which are priced and marketed separately. In the case of the HP Integral Personal Computer, the subsets are necessarily much smaller, much like LSX, since only diskettes are available for secondary storage. To get around the limited secondary storage available however, the kernel was supplied in ROM and part of the RAM was available for temporary files.

5. Personal UNIX System Trends

One can look back over the last ten years and see how technology changes in both hardware and software have enhanced the capabilities of the personal UNIX system. However, it is much more difficult to predict what will happen over the next ten years. Thus we will focus on the near term, that is the next three or four years, where the technology trends are more predictable and discuss how these will impact the functionality available to the personal UNIX user.

UNIX systems will continue to fill the need of PC users who require multi-tasking, even though there are fewer application programs available under the UNIX operating system than there are under the MS-DOS system. The number of applications available for UNIX systems is increasing dramatically, as one can see by consulting the various Catalogues of Software available for the UNIX system. Once a critical mass of systems has been achieved, more and more application developers will build applications for UNIX systems. Applications which today run on minicomputers and mainframes will be ported to run on microcomputers as well. Networking will be used to tie all of these computers together, giving the user a uniform interface regardless of which system he is using.

5.1 Software Technology Trends

Software running on personal computers is very sensitive to standardization trends. The current standard interface for UNIX application programs is the System V Interface Definition defined by AT&T, but influenced by the /usr/group Standards activity. It is expected that the industry will follow this trend, and therefore make the job of the applications vendor easier. There are a number of key areas in the UNIX system that will receive a lot of attention over the next few years and hopefully evolve into standards. These include the areas of networking, distributed file systems, graphics, and bitmap terminal support.

Networking is required to tie the personal computers into minicomputer and mainframe networks for the office environment. For home-based personal UNIX systems, remote dial-in access is sufficient. Distributed file system support is convenient but probably less important for personal computers in the office since dedicated response is more important. Connection to file servers and other network services will be important. To support high performance graphics a dedicated personal computer with a bitmapped terminal is required. This is also required for interactive text processing applications, and especially for WYSIWYG word processing systems.

There will be a continuing demand for better user interface software, utilizing window managers. This will require good window management software, with graphics support and complementary job control facilities. Good interprocess communication facilities will continue to play an important role in improving the interaction of various application programs, and in improving the response of the system.

In the operating system itself, one will continue to see new features being added to support the needs of the user and to improve performance. The use of shared libraries will become more

common as the need to conserve disk space, speed up program load time, and minimize memory usage, increases. For personal UNIX systems in particular, loadable device drivers will be required by the fact that the UNIX system itself will be located in ROM, to promote the trend to compact and portable personal UNIX systems.

Finally it is expected that there will eventually be a trend away from procedural languages to speed up the development of new application packages. Fourth generation languages will become more prevalent, offering software developers the benefits of both structured and functional programming for expert systems, logic programming, computer-aided design and databases. Expert systems are being developed now and we are already starting to see the introduction of application generators.

5.2 Hardware Technology Trends

It is expected that we will be seeing faster and more powerful microprocessors being used for personal UNIX systems in the next several years. The 68020, 386, 32032 and Z80000 are all powerful 32-bit microprocessors that will be used in personal UNIX systems. AT&T has recently introduced its 32-bit microprocessor for use in the commercial marketplace as well. This will lead to more sophisticated application packages with better response times. In addition, the next year will see the widespread use of 1Mbit memory chips, to be followed by even larger capacity in a few more years. This will enable one to have a much more complex and powerful personal UNIX system (What happened to small is beautiful?) capable of supporting higher resolution color displays with interactive graphics. Graphics, bitmapped terminal support and networking are all likely to find their way into the UNIX kernel very soon. Window managers with their improved user interface characteristics will become more prevalent. Screen editors will become the norm, rather than the exception.

The prices of ROM, EPROM and EAROM are dropping and will become price competitive with diskettes. They may eventually replace the diskettes in certain applications. They of course have a much better access time than diskettes. Winchester disks are expected to be standardized at the 3.5" size soon. Their capacity is expected to grow significantly with new recording techniques. Removable winchester disks should replace the fixed winchester disks over time. This will help with the backup situation and of course result in physically more compact and portable systems.

Another relatively new technology is expected to impact the use of personal UNIX systems soon. The laser disk technology is maturing, allowing one to store hundreds of megabytes of data for quick access by a personal computer. These are eventually expected to store images, sound and motion pictures, and will have wide applicability in the education and training fields.

The bitmapped terminal is still a good match for a personal computer dedicated to one user. The increase in memory size and the high bandwidth path to the terminal from the CPU memory will make color graphics, interactive text processing, and WYSIWYG word processing inexpensive to support. The high bandwidth compared to that of asynchronous terminals in earlier days, makes the bitmapped terminal ideal for the single user since performance is not significantly degraded. But for compact personal computers, CRT technology is probably close to reaching its limit. LCD displays are becoming more economical, and they have the advantage that they require low power, use integrated VLSI circuits and therefore are very suitable for portability.

The personal UNIX computer user needs to access data on other computers. For home computers, 1200 baud modems are the norm today. But 2400 baud modems have been introduced and will become increasingly popular. For the office user, Ethernet connections are available, but quite expensive. IBM has introduced the 2 Mbit broadband PCNET, while AT&T has introduced the 1 Mbit baseband Starlan network. It is expected that these two networks will become standards in the personal UNIX systems market because they are well suited to the data rate requirements, and improved technology in LAN controllers will reduce the access costs to the network.

In summary there are still factors of improvements available in the technology of the components that make up a personal UNIX system. Thus we will continue to see performance gains and some functionality improvements in the software. While the cost of a persona! UNIX system is now $5000 to $6000 dollars, the cost is not likely to decrease much over the next few years. Rather the decrease in the component costs will be matched by increased capacity of the system and increased functionality gains in the software.

Parlez-Vous L'UNIX†?
The European Perspective, Past and Future

Jean Wood

Digital Equipment Centre Technique European SARL
Valbonne, France

Hans-Joachim Brede

Digital Equipment GmbH
München, Germany

ABSTRACT

One drawback of living in Europe is that the USENIX Pre-Announcement and Call for Papers is received after the deadline for abstracts has passed. Fortunately, there are more interesting aspects of UNIX operating system usage in Europe, as well. Timezones, language and nationalism create obstacles to traditional interaction between supplier and customer. Electronic communication is governed by nationalised PTT's and priced to make traditional UNIX networking difficult. European character sets and other language dependencies create new problems in the man/machine interface. These variations on communication in a European UNIX system environment are discussed, as well as the resulting frustrations and innovations.

1. Introduction

This paper offers some insight into the special challenge of using UNIX software in Europe. It describes the ways that communications between people, between machines and between people and machines are affected by the environment. It is not an exhaustive report on multi—national character sets, the hardware and software that support them, or other national language issues. It does not provide *THE SOLUTION,* or even suggest that *THE SOLUTION* exists. Hopefully it will increase the awareness of some of the issues raised and stimulate interest in new areas to be considered in future UNIX system development.

2. Character Sets (Man/Machine Communication)

The existence of different language character sets in Europe and the implications of the differences are acknowledged by most American developers and manufacturers. Many have some knowledge of French, German or Spanish or have been exposed to the occasional European word adopted or adapted into American English usage, such as Hägen Daz. *Surviving with seven bits.* The Unix system supports the 7 bit ASCII code which defines a character for each of the 128 bit combinations. Most European languages require some additional characters. In German, for example, there are additional characters called umlauts and the sharp s. which can be produced with nroff using the escape sequences in Table 1.

---

†UNIX is a trademark of AT&T Bell Laboratories

Table 1

ä	a*:
Ä	A*:
ö	o*:
Ö	O*:
ü	u*:
Ü	U*:
β	\(*b

It is not reasonable to expect a German secretary or other non-technical user to type a standard German text, using sequences such as these to create what are to him normal alphabetics like those found on a local typewriter. Many European terminal keyboards use national replacement character sets which include these additional characters by sacrificing some others which are near and dear to the heart of many a C programmers: [,], {, }, ¦, and \\. Because they have the same ASCII values as the characters they replace, UNIX utilities have no problems with C source programs and shell commands which may look a little unusual to the North American programmer:

```
ä
        int aÄ20Ü;
        .
        .
        .
ü
```

Many users of such keyboards add the necessary labels to make it easier to remember the list of equivalent symbols or order replacement 'American' keyboards. The C compiler and the Shell quite easily accept input containing the replacement characters, and interpret the additional alphabetics as operators, punctuation, etc. In fact the file(1) program will often identify European text as 'C source' as does the C compiler., Not too surprising until it actually tries to compile it. Unfortunately, programs expecting alphabetic input are not as accommodating. Because text is sorted according to ASCII value, all words containing umlauts et al. do not sort correctly. Hyphenation rules do not work well on words which contain what appear to be punctuation or other non-alphabetic characters instead of vowels. *Moving to eight bits.* Moving to an eight bit character representation only solves the easy problem (unique character representation); there are others. Collation and hyphenation rules are language dependent and may not be consistent even within a given language. For example, in Germany there are two ways of sorting, one that is used by the telephone company (which treats the umlauts as a vowel followed by e) and one used by the rest of the German speaking world, which treats the umlauts as unique vowels. The former method (ae for ä) is often used to represent umlauts on ASCII terminals and printers. In Spanish some double letter combinations are treated as a single letter for purposes of sorting. Hyphenation may change the spelling of a word as in the German Kuckuck (cuckoo), which, when hyphenated becomes Kuk—kuck. Capitalisation can also affect the spelling of a word. French characters with accents may or may not retain them when they are case shifted depending upon whether you are in France, Belgium, Switzerland or Canada. Guist and Missimer [1] developed the following list of language dependent parameters:

1. Character Size. Expanding to 8 or 16 bit character size means that software that edits strings must know how many bytes make a character; and must not use the eighth bit as a flag (as sh, csh and vi freely do.)
2. Case shifting and awareness of spelling changes.
3. Collation must vary according to the rules of the language of the source and the purpose of the collation.

4. Directionality of displayed text (left to right, right to left, horizontal, vertical, etc.; happily not an issue in Europe)
5. Classification of characters on the basis of coded value can no longer be hard-coded.
6. Escape sequences used to switch between character sets in a multi-language environment.
7. Hyphenation and spelling.
8. Date and time computation and display.
9. Days of the week and month.
10. Currency units and the subdivision thereof.
11. Representation of numbers, support for different radix character symbol, variation of symbol used for grouping and number of digits grouped.
12. Error messages, prompts and responses to prompts.

Once an *international* system exists, there will be new problems. Consider USENET or EUNET and connections to many other less international systems. Software in these potentially heterogeneous networks must be able to handle the translation of data between 7, 8 and 16 bit character representation and 'do the right thing'. There are other problems in communications between 'international' systems and traditional ones such as those encountered by Siemans when they translated all of the text of their system into German. Most mailers could not understand the 'Absender' and 'Von' lines in the header; these were changed back to 'From' and 'To'.

Localisation. Localisation is the word often used to describe the current approach to providing support in the UNIX system for European languages [2]. By definition, the word refers to a confinement or restriction to a particular area or part; it was historically used by the English in India to describe one of their objectives there. Because of the negative connotations, many people prefer the more cumbersome (and difficult to spell and pronounce) term *Internationalisation.* Native language support of this kind generally refers to translation of error and other messages and correct representation of numbers, dates and time. Additionally, it could include translation of header files, appropriate language dictionaries, hyphenation rules for text processing and translation of fortunes. During development planning, consideration must be given to how many languages will be supported concurrently. At the EUUG Spring, 1985 Conference in Paris, an AT&T representative suggested that the support of 2 languages at a time would suffice (perhaps English and one other?) Another approach, such as that used by DEC[1] in their set of VAX/LOLA products (Local Languages support for VAX/VMS) allows the addition of support for as many languages as required (each LOLA language kit is purchased individually), and provides a mechanism for users to set and change their environment as needed. A major problem in any endeavour of this nature is the lack of perspective. Correct implementation of language dependencies is best accomplished by a native; coordination of such an effort among several groups while maintaining compatibilty is non-trivial. *Putting the problem into perspective.* Consider Greece. The Greek language does not require a few extra characters, but an entirely different alphabet; a much more complicated alphabet than those typically discussed in papers on internationalisation. A case study conducted at the Research Center of Crete describes the significant characteristics of Greek and an approach for support for the language in the UNIX system [3]. Besides a unique character set, there are two different and non-compatible systems of diacritical marks, some unique punctuation, and two forms of the lowercase symbol sigma. 'Teliko sigma' is used when sigma appears at the end of a word; the regular sigma is used in all other cases. Although the two forms are displayed differently, internally they should be treated identically. There are additional characters in 'classical Greek', as well as a letter based numbering system, similar to Roman numerals, called Byzantine numerals. In Japan and China the level of complexity is even greater; don't French and German seem a lot easier to cope with now?

1 1 VAX, VMS and ULTRIX are trademarks of Digital Equipment Corporation

3. Networks (Machine to Machine Communication)

Whenever more than one UNIX system are within a few kilometers of each other, the appeal of setting up communications between the two is overwhelming. The process in Europe is almost always complicated by the involvement of one or more PTT's. *The Bureaucracy of the PTT.* The postal/telephone/telegraphic services are nationalised entities controlling nearly all forms of communication (ignoring the odd verbal exchange). All versions of all communications software and hardware must be certified by them. PTT regulations in France, for example, require that an autodialer contain a prerecorded announcement that is played if the dialer reaches a person instead of a machine. Modems of this type are in use at Digital Equipment's European Remote Diagnostic center in Valbonne, France. This facility provides remote diagnostic and monitoring services for all of DEC's customers in continental Europe; it is just as likely that the autodialer will be 'speaking' French with an unsuspecting Swede, Italian or German as with a French person. *European Innovations.* Most UNIX systems do not support European PTT autodial modems which are, at any rate, quite rare and expensive. Given poor quality telephone lines, modem compatability problems and the associated financial costs, it is easy to understand the move to other forms of communication links for UUCP and other connections. R. Linhart [5] researched the options available for UUCP connections in Germany and their relative performance and cost. Table 2 provides a brief overview of common German public networks.

Common Public Networks

network type	typical speed	mean bit failure	international availability
telephone	1200 async	2*10^-4	low (few autodialers)
leased lines	9600 sync	10^-6	high
X.21	9600 sync	10^-5	low
X.25	9600 sync	10^-9 (HDLC)	high

Table 2

X.25 services not only have the highest reliability at reasonable speeds, but are also much less expensive. Relative cost is presented in Table 3 in U.S. dollars/Mb.

Average costs USA <-> Germany

	phone (1200 Bd)	
USA domestic	25..65 (coast to coast)	4..15
Germany domestic	15..65	4..20
USA -> Germany	110..130	190 (+$10/hour)
Germany -> USA	300..360	90

Table 3

Monthly fees are much higher in the U.S. ($1500 vs. $200) for X.25 lines; there is a monthly charge in Germany for the modem for telephone connections ($45). Linhart found that compared to raw leased telephone lines in Germany, X.25 packet switched networks are up to:

- 8 times cheaper
- 8 times faster
- 100,000 times more reliable

In Europe, these X.25 links are fast becoming the standard for UNIX to UNIX system connections. They are used by more than half the USENET sites in France and more than 90 percent of those in Germany. There is still a need for real X.25 support in the UNIX software to enable connections to non—UNIX based machines. Open System Interconnect protocols should be implemented on X.25. *Other European Specific Networking Needs.* Understandably, there is little interest in UNIX development organisations in the United States in creating software to

communicate with Siemens or Honeywell Bull systems. This lack of interest on the part of American developers and marketting groups does not diminish the European need for such products.

4. Reach Out and Touch Someone: (Man to Man Communication)

Until recently, one of the most difficult forms of communication known to European UNIX system users was reaching an 'authority' at AT&T or Berkeley. The basis for the problem was not a lack of a common language, but the inconvenience of waiting 5 to 10 hours for someone on the other side of the world to turn up at the office. If the person being sought was of a technical nature, then the window shifted, based on their nocturnal habits. If the reason for communication was of a licensing nature, the European licensing expert had to be persuaded to wait as well. *Establishing European Operations (Local Communication)* Now there is UNIX Europe, Ltd. (UEL) in London representing AT&T/Olivetti; DEC and Hewlett Packard have UNIX trained European support people and Pyramid, Sun Microsystems, Masscomp and others have begun European operations. It is unclear whether these new developments have solved more problems than they have created. Excerpts from an EUNET discussion [6] of UNIX EUROPE, Ltd:

> It's nice to have place where you can actually phone during working hours. Also, it is not so far away, so phone rates are reasonable. And sometimes a phone call is necessary ... I generally find the British easier to talk with in matters of business... NO. NO. NO. NO. NO. NO. It is NOT better with UNIX EUROPE. I sent them a letter and requested a license for one more CPU it was sent from me 15 October 1984, I received a letter from them 11 December saying they was sorry for the long delay and they would deal with my aply soon. I have not received any further letters from them. They claim it is because of a large backlog of educational requests, but 4 month thats worse than in US. (In response to a request for UEL address/phone information) Try 'cu /dev/null', has about the same effect:-)

A year ago a Swiss DEC salesman denied the existence of any Digital UNIX product to a potential customer who had just returned from a USENIX conference where he had seen and 'touched' ULTRIX. When faced with the proof, the salesman still refused to sell a VAX without VMS. Some habits die hard. Similarly many HP salesman preferred to demo BASIC systems instead of HP/UX[2]. Through recruiting and training efforts, fortunately, this behavior is changing.

European Standards (Communications in Committee). As the UNIX operating system evolved into it's many variations, the need for a well defined standard became evident. Late in 1984, in addition to standardisation activities in the U.S. and Japan, five European computer manufacturers began efforts to define a standard UNIX. This group is known as BISON, taking the first letter from each of the founding members: (Honeywell) Bull, I.C.L., Siemens, Olivetti and Nixdorf. They were joined earlier this year by Phillips. A number of other computer manufacturers, including Hewlett Packard and Digital Equipment Corporation have expressed interest in participating in the BISON standard definition process. Despite the size of their respective European subsidiaries or the existence of their European engineering groups, they are not considered eligible because both companies have corporate headquarters in the U.S. and are, therefore, not European. The goal of BISON is to define a UNIX standard, from the ground up, suitable for the European market. This redefinition process is reminiscent of another effort to design a better mousetrap, the SOL[3]. Project. The SOL project, launched by l'Agence de l'Informatique and carried out by INRIA with the participation of the CNET developed a portable basic software engineering environment, which included Pascal compilers, a timesharing operating system, adapted to (French) mini and micro computers, which was 'totally compatible with UNIX' [4], and a set of basic system utilities and software tools well known to most UNIX users. The reasons for this 're-development' of UNIX were a commercial or strategic independence from AT&T licensing and a technical independence from DEC hardware architecture. One of the major

2 HP/UX is a Trademark of Hewlett Packard

3 SOL is a registered trademark of Agence de l'Informatique.

differences between UNIX and SOL is the development language; SOL is written in an extended form of Pascal.

5. Minor Irritations (or no communication at all)

It is the insignificant, easy to fix things and the ones that continue to be done wrong that are the most exasperating. *Time.* Everyone knows there are different timezones in the world and that the sun rises in the east and sets in the west. There is some confusion about what GMT and the international dateline are. (They are NOT the same thing.) If it is June 13 7:00 BST (British Summer Time) in London, then it is June 13 8:00 MET (Middle European Time) DST in München or Valbonne. At the same time it is June 13 2:00 EDT and June 12 22:00 PDT. (Time is one of the areas in which Europeans are ahead of the U.S.) To describe MET in a system configuration file, the +5 value (EST) is changed to -1. No problem, until ULTRIX Engineering created a simple—to—use, automatic installation procedure which asks you to specify your timezone in hours ahead (meaning west) of GMT and only allows a positive response. When asked for a recommended action for Europeans, the suggestion from Merrimack, New Hampshire was to use the value 23. But that means crossing the date line with the result of June 14 9:00 BST occurring at the same time as June 13 8:00 MET, instead of June 14 10:00 MET as discussed above. It is confusing, isn't it? All the ULTRIX systems on the Continent aren't really 26 hours behind, though, because it is still possible to edit the configuration file and insert the desired '-1'. But shouldn't the friendly, simple—to—use, automatic installation procedures work everywhere in the world? *Saving Time.* Another timely issue is Daylight Savings Time. This year all Berkeley based systems in Germany switched to Daylight Savings Time one week before the government did. It is not possible to convince the governments of European countries to follow the programmed algorithms. Politicians go their own way; perhaps UNIX software can follow them. *Saving Money (International Unix User Associations).* /usr/group calls themselves 'International' of UNIX users. It would seem to be an interesting group for a European UNIX User to join. A check in U.S. dollars (drawn on a the New York branch of a German bank) was sent to /usr/group along with a membership form on October 23, 1984 (23. Oktober 84). Five months later the check was returned with a form letter with the appropriate boxes checked and a hand written note requesting that the request be resubmitted with an international money order with magnetised numbers on the bottom.

6. Conclusions

A wide range of problems must be solved in the design and implementation of a European UNIX System, from simple hacks to large scale engineering projects. Awareness of the issues is spreading among the manufacturers and developers in the U.S. who plan future software releases. Hopefully they will also recognise the value of European involvement in such plans.

Acknowledgements

Many thanks are due to the members of the European UNIX community who have shared with us their always entertaining experiences including: Jaap Akkerhuis, Peter Collinson, Teus Hagen, Radek Linhart, Jim McKie, and Hans W. Strack—Zimmermann.

References

[1] J. Guist and D. Missimer, *What is an international UNIX system?,* Proceedings European UNIX Users Group Conference, Spring 1985.

[2] P. Tintel, *EURIX - a UNIX based system using European natural languages,* Proceedings European UNIX Users Group Conference, Spring 1984.

[3] S. Hull, *Greek Characters on UNIX* Proceedings European UNIX Users Group Conference, Spring 1985.

[4] *Club Sol Info, SOL Users' Information Letter* English special Issue No. 1, September 1983.

[5] R. Linhart, *UNIX networking via X.25* EUUG Newsletter, Fall, 1985.

[6] T. Hagen, D. Karrenberg, J. McKie, G. Olofsson, and H. Albertsson, contributions from USENET groups

An Exception Handler for C

May 1985

Eric Allman
David Been

Britton Lee, Inc.
1919 Addison, Suite 105
Berkeley CA 94704

ABSTRACT

Large systems often need to handle exceptional conditions such as errors or interrupts. Exceptions usually require a change in the flow of control that does not fit neatly into the usual "structured" control structures. Various types of non-structured transfer statements (such as *signals*, *setjmp/longjmp*, or *goto*) are usually used instead; these are subject to serious problems such as transfers to inactive contexts, inscrutable code, and failure to release resources.

Exception handlers provide a structured non-local transfer capability. When an exception is *raised* execution of the current procedure is suspended and an appropriate handler is invoked. The handler may elect to *continue* processing, returning after the raise like a normal subroutine, *back out*, aborting the current procedure and any others back to the procedure that set the handler, *re-raise* the exception, passing it back to a previous context, or *terminate* the program.

In general, exception handlers assist the programmer to separate the identification of errors from the resolution of the error (that is, "I have detected an exceptional condition that should be handled" versus "when you notice this condition, perform this action"). The resolution of exceptional conditions can be dependent on the context in which they are raised.

An exception handler for C is described. It requires no compiler changes. Only minimal knowledge of the runtime environment is required. This knowledge is embodied in two assembly-language routines comprising approximately fifteen lines of code on most processors. The remainder of the implementation is machine-independent, relying on the *setjmp* and *longjmp* routines to provide non-local control flow.

INTRODUCTION

A common difficulty in programming non-trivial systems is handling exceptional conditions such as errors or interrupts. Two serious problems present themselves. First, exceptions are inconvenient to test for in all possible situations. Much UNIX† code *assumes* that common library calls will succeed; this has been the cause of many latent bugs [Darwin85]. Second, exceptions usually require a change in the flow of control that does not fit neatly into the usual control structures. Some form of the goto statement is the most common recourse. Although the problems of

†UNIX is a trademark of AT&T Bell Laboratories.

the **goto** statement have been well documented [Dijkstra68], their utility for handling exceptional conditions has been defended [Knuth74].

The UNIX C environment contains three separate facilities related to handling of exceptional flow of control: the **goto** statement, *setjmp* and *longjmp* (non-local **goto**), and signals (asynchronous exceptions). These are often used in conjunction with one another. For example, when a signal is caught that should interrupt processing the signal handler will typically use longjmp to transfer control to a top loop.

Although adequate for simple applications, this can create problems in more complex systems. For example, consider the problem of releasing resources (e.g., file descriptors) when an interrupt is caught. This must normally be handled using global variables, e.g.:

```
sigcatch()
{
        extern int somefd;
        extern jmp_buf top;

        if (somefd ≥ 0)
        {
                close(somefd);
                somefd = -1;
        }

        longjmp(top);
}
```

In general global variables create more problems than they solve, making this an unattractive solution.

An exception handler provides a structured solution to this problem. When an exceptional condition occurs an exception is *raised*, suspending execution of the current procedure. A search is made back through the runtime stack for a context that wishes to *handle* the exception. Control passes to the handler. The handler may decide to *continue* (that is, continue processing after the raise) or *back out* (that is, abort the current procedure and any others back to the procedure that set the handler).

In general, exception handlers let the programmer express *intent* ("I have detected an exceptional condition that should be handled") rather than *implementation* ("when you notice this condition, perform this action"). This makes code less context dependent, thus more portable and more reusable.

Since all errors can be handled consistently (rather than possibly returning zero or minus one, raising a signal, printing a message, etc.) programs are easier to understand. Since the programmer must specifically ignore exceptional events, unexpected failures are caught quickly; latent bugs are less likely and their identification is easier. Since errors are events rather than global values, status handling and reporting in complex applications (such as libraries built on top of other libraries) becomes easier and more accurate.

A substantial overview of exception handlers is presented in [Goodenough75]. That discussion presumes that the exception handler is integrated into the programming language.

Section 1 describes the semantics of our exception handler. Section 2 describes how we implement this package without requiring language changes. Portability of the package is described in section 3. An analysis of the exception handler, including a comparison to exception handling facilities in other languages such as Ada, is given in section 4. Section 5 gives some figures describing the performance of exceptions.

1. SEMANTICS

This section describes the semantics of the exception handler.

1.1. Exception Identification

Exceptions are identified by a character string. Since these are simple strings, their scope is completely global. By convention exception identifiers are dot-separated sequences of words identifying facilities.

Strings allow us to handle exceptions on the basis of patterns rather than unique exceptions.

EXAMPLE 1: ▽

The pattern:

E:IDMLIB.IO.*

will match any error exceptions raised by the *IO* subsystem of the *IDMLIB* library. △

Encoded into the exception identification is a *severity* code. The severity gives clues for how to handle exceptions. For example, severities can be Abort to indicate that the procedure raising the exception cannot continue, Error to indicate that the ultimate results of this routine will probably be wrong but that processing can continue, or Warning to indicate that the condition has been resolved. The severity is not actually considered part of the identification; for example, the exceptions "E:X" and "A:X" represent the same exception with different severities (for example, when an error in one context is fatal in another).

EXAMPLE 2: ▽

The pattern to handle all abort or error conditions is:

[AE]:*

△

Severities may have some semantics built into the exception handler. For example, Abort severity exceptions are never allowed to return to the procedure that raised them. This corresponds to Goodenough's ESCAPE exceptions and to all Ada exceptions. Programs may modify these semantics. For example, a handler may catch Errors and not return, making them behave as though they were Abort exceptions.

1.2. Basics (Handle/Raise)

1.2.1. Handling exceptions – exchandle

A procedure may indicate a desire to handle a particular exception by issuing the *exchandle* call:

int exchandle(*exception_pattern, handler_procedure*)

The *exception_pattern* is a character string that indicates the exceptions that will be caught and processed by the *handler_procedure*.

When *exchandle* is called it returns zero.

EXAMPLE 3: ▽

To send all error conditions to the routine *mark_error*:

exchandle("E:*", mark_error);

△

When the procedure that set a handler returns that handler is automatically removed.

1.2.2. Raising exceptions – excraise

A procedure may raise an exception using the *excraise* call:

excraise(exception, arguments ...)

Excraise searches backwards through a runtime stack of exception contexts for an *exchandle* call with an *exception_pattern* call matching the raised *exception*. When found, the *handler_procedure* specified by *exchandle* is called; the *arguments* of the *excraise* call are available to it.

EXAMPLE 4: ▽

To raise an exception with no arguments:

```
excraise("E:MYEXCEPTION", NULL);
```

△

If no handler for an exception is found a default action occurs. Default handlers are described in more detail below.

1.2.3. Exception nesting

If a handler for an exception cannot be found in the current context then the enclosing context is searched. See below for an example.

When a context is destroyed (i.e., when a procedure returns) any handlers in that context are removed. This guarantees that a non-local **goto** to a context that is no longer active is impossible.

EXAMPLE 5: ▽

Consider the code:

```
a()
{
        extern g();
        exchandle("?:X", g);
        b();
        excraise("W:X", NULL);          /* 1 */
}

b()
{
        excraise("W:X", NULL);          /* 2 */
        c();
        excraise("W:X", NULL);          /* 3 */
}

c()
{
        extern h();
        excraise("W:X", NULL);          /* 4 */
        exchandle("?:X", h);
        excraise("W:X", NULL);          /* 5 */
}
```

In this example the *excraise* calls labeled 1, 2, 3, and 4 are all caught by handler *g*, while the *excraise* call labeled 5 is caught by handler *h*. The *excraise* labeled 3 is

not handled by *h* because the handler set in *c* is removed automatically when *c* returns. Note that the call labeled 4 is passed back through two contexts automatically.

Δ

1.2.4. Arguments

Arguments may be passed to the exception handler from the *excraise* call. Because the C runtime environment does not support tagged types, all arguments are constrained to be of type pointer-to-**char**. The argument list is terminated by a NULL pointer.

EXAMPLE 6: ▽

To raise an exception passing two arguments:

excraise("E:X", "arg_1", "arg_2", NULL);

Δ

1.2.5. Exception handler procedures

The *handler_procedure* can perform any operations it wishes *except* for executing a non-local **goto** (such as a *longjmp* call). It then has three options for *disposition* of the exception:

- Return zero. This will return control to the *excraise* call so that the procedure detecting the exceptional condition may continue. This matches Goodenough's RESUME primitive.
- Return non-zero. This will return control to the *exchandle* call that placed the handler. Any procedure contexts between the context that placed the handler and the context that raised the handler are unwound automatically. This is the structured equivalent of the non-local **goto**; it is similar to Goodenough's EXIT primitive.
- Raise another exception. If the same exception is raised (a "reraise" operation) control passes further back through the stack, that is, it will be caught by an enclosing handler.

The arguments passed to *excraise* are given to the handler in the same manner as arguments to a UNIX command; that is, they are collected together into a vector of strings (normally named *excv*). *Excv[0]* is the name of the exception that was raised.

EXAMPLE 7: ▽

An exception handler that prints the exception code and then continues:

```
myhandler(excv)
        char **excv;
{
        printf("%s\n", excv[0]);
        return (0);
}
```

Δ

1.2.6. Semantics of raising exceptions inside a handler

Handler procedures run as sub-contexts of the procedure raising the exception. Hence, any exception raised inside a handler is processed relative to the raise call, rather than relative to the handle call.

EXAMPLE 8: ▽

Consider the code:

```
a()
{
        extern g(), h(), i();
        exchandle("W:X", g);
        exchandle("W:Y", h);
        exchandle("W:Z", i);
        b();
}
b()
{
        excraise("W:X", NULL);
}
g()
{
        extern j();
        exchandle("W:Z", j);
        excraise("W:X", NULL);
        excraise("W:Y", NULL);
        excraise("W:Z", NULL);
}
```

When W:X is raised procedure *g* will be called. Inside *g*, the raise of W:Z will (obviously) be handled by handler *j*. Exception Y will be handled by handler *h*. However, exception X is passed back past procedure *a* because handlers are temporarily disabled while they are being processed so that another raise of the same exception will not cause loops. For example, the *exchandle("W:X", g)* in *a* is temporarily disabled while *g* is executing. Δ

1.3. Asynchronous Signals (Alock/Aunlock)

UNIX signals are integrated with the exception handler[1]. Signals are translated to exceptions with the special severity Transient (to indicate that they will probably not occur in the same place if the program is re-run). For example, keyboard interrupt (the SIGINT signal in UNIX) is translated to T:IDMLIB.ASYNC.INT (where IDMLIB is the library containing the exception handler).

In many instances asynchronous events must be masked out (e.g., when modifying global data structures) in order to prevent inconsistencies. The routine *excalock* will temporarily mask out asynchronous events. *Excaunlock* is the reciprocal routine, unlocking asynchronous events and processing any interrupts that occurred in the interim.

Excalock and *excaunlock* calls nest, that is, two lock calls followed by a single unlock call do not unlock interrupts. This allows procedures that must be atomic to utilize *excalock/unlock* without concern for the environment.

Since backout is possible even when asynchronous events are locked (e.g., by an explicit exception raise), *excaunlock* takes a Boolean argument *force*. If *force* is FALSE *excaunlock* operates as described above, but if TRUE all *excalock* calls are undone and asynchronous events are processed immediately.

EXAMPLE 9: ▽

To update a data structure in a critical section:

[1] UNIX signals have nothing to do with Goodenough-style SIGNAL's.

```
excalock();
/* update the data structure */
excaunlock(FALSE);
```

Δ

1.4. Default Actions (Dhandle)

If, when an exception is raised, no handler has been instantiated (that is, no *exchandle* call with a matching pattern has been found) a *default action* is performed. The normal default action is to print a message associated with the exception. Abort severity exceptions then cause process termination; others cause the *excraise* to continue.

Users can override default actions using *excdhandle*. This routine is identical to *exchandle* with the following modifications:

- Default handlers are searched after all non-default handlers, regardless of the order in which they were instantiated.
- Default handlers may not try to backout. An attempt to do so will abort the program.
- Default handlers are not automatically removed when the routine that set them returns.

EXAMPLE 10: ▽

Consider the code:

```
a()
{
        b();
        excraise("W:X", NULL);
}

b()
{
        extern int g(), h();
        exchandle("W:X", h);
        excdhandle("W:X", g, NULL);
        excraise("W:X", NULL);
}
```

The *excraise* call in *a* will cause *g* to be invoked as the handler. However, the *excraise* call in *b* will invoke *h* as the handler. Δ

Of course, "real programmers" can simulate regular UNIX semantics by ignoring all exceptions by default.

1.5. System-Provided Handlers (Print/Abort/Ignore/Backout)

Several handlers are provided by default owing to their general usefulness.

Excprint prints out the text of a message associated with the exception and continues (that is, returns zero so that the *excraise* call will return).

Excabort modifies the exception to an Abort severity exception and reraises it. This allows another handler to process the exception, but restricts that handler from returning to the *excraise* call. An attempt to do so will abort the program.

Excignore silently ignores the exception.

Excbackout silently backs out from the exception.

1.6. Miscellaneous Routines (Vraise)

The routine *excvraise* is identical to *excraise* except that it takes the exception arguments as a single vector rather than a list of arguments in the function call.

1.7. Cleanup on Backout

Routines that allocate resources (e.g., memory or open files) must be able to regain control during an exception backout that will abnormally terminate the routine. A special class of *cleanup handlers* is provided for this functionality.

Cleanup handlers are instantiated using *exccleanup(func, arg)*. If the handler decides to backout the context stack is searched a second time for cleanup handlers.

EXAMPLE 11: ▽

To release resources if backout occurs:

```
a()
{
        if (exchandle("[AE]:*", excbackout) == 0)
                b();
}

b()
{
        IFILE *ifp;
        extern ifclose();

        ifp = ifopen( ... );
        exccleanup(ifclose, ifp);
        process(ifp);
        ifclose(ifp);
}

process(ifp)
        IFILE *ifp;
{
        excraise("E:XYZZY", NULL);
}
```

The handler corresponding to the *excraise* call is the one instantiated in *a*; when the backout occurs the cleanup handler instantiated in *b* will be called so that the file will be closed. △

2. IMPLEMENTATION

One *exception context* is created for each procedure context that sets any handlers (i.e., invokes *exchandle*). The exception context (essentially an expansion of the language runtime context) contains the actual return address for the procedure (as will be described below) and a list of *exception vectors*. Our implementation of the exception handler maintains the exception contexts on a private stack. If the exception handler were integrated into the language the runtime stack could be used.

Each exception vector corresponds to an *exchandle* call. The vector contains the pattern describing the exceptions that this handler will catch, a pointer to the handler function, the argument it should receive, and a "jump buffer" containing the context needed to return again from the *exchandle* call in the event that the handler decides to back out.

The standard UNIX *setjmp/longjmp* primitives are used to store the backout context (in the form of a jump buffer).

2.1. Exchandle — Setting a Handler

The routine *exchandle* is actually a macro:

```
#define exchandle(exc, fn)  setjmp(_excvect(exc, fn)→_exc_jb)
```

The internal routine *_excvect* performs the following actions:

- Get the return address of the procedure calling *exchandle*.
- If this is the first time that a handler has been instantiated in the current procedure context, allocate a new exception context. Save the return address in the exception context, and replace the return address in the procedure context by the address of the pseudo-routine *_excdisable*[2].
- Create the vector for the exception instantiation, storing the *exc*, *fn*, and *arg* information in it.
- Return the exception vector. The jump buffer in the exception vector is then passed to *setjmp* by the *exchandle* macro, thus saving the context necessary for backout.

2.2. Excdisable — Clearing a Handler

When the procedure that set the handler returns, it runs through the usual procedure epilogue and then jumps to the return address in its stack frame. However, this return address has been replaced by the address of *_excdisable*. This is a small assembly language assist (which is *not* a subroutine!) that performs the actions:

- Save the return value from the function that is returning.
- Call the routine *_excdrop* to pop the exception context stack. *_Excdrop* returns the actual return address as its value.
- Save the return value from *_excdrop* (which is the actual return address that we should ultimately transfer to).
- Replace the return value from the function we are cleaning up after.
- Jump to the return address returned by *_excdrop*.

These steps can be performed in about ten to fifteen lines of assembly code on most architectures.

2.3. Excraise — Raising an Exception

Raising an exception performs the following steps:

- Search backwards through the exception context stack for an exception vector with a pattern matching the exception being raised. Vectors with NULL function pointers are ignored for this search, for reasons described below. (The user-supplied default handlers are linked into the bottom of the context stack, so no special search of the default context need be made.)
- If no handler has a pattern matching the exception, print the exception and set the disposition of the exception to "continue."
- Otherwise, call the handler specified in the exception vector. However, save the address of the handler function and replace it by NULL while the handler function is being invoked, restoring it immediately upon return. This prevents reraises of the same exception from invoking the same exception handler. The return value of the handler function becomes the exception disposition.
- If the disposition is continue and the exception was not an Abort severity exception, return (from the *excraise* call).
- If the disposition is continue but the severity is Abort, print a message and abort the program.
- Deallocate any exception contexts between the current context and the context containing the exception vector that we are processing. These correspond to contexts that will no longer be active after we back out. As we deallocate the contexts, process any

[2]If the return address is not *_excdisable* this must be the first handler instantiated in this context.

backout handlers we find.

- (The exception context stack is now rolled back to what will soon be the current context.) Deallocate any exception vectors that resulted from *exchandle* calls later in the procedure setting the handlers than the *exchandle* call that we will return to. These handlers will be set again as the procedure reexecutes.
- Perform a *longjmp* back to the context that originally set the handler. The *longjmp* will clean up the runtime stack and transfer control.

3. PORTABILITY

The implementation of the exception handler is highly portable between UNIX systems. Use of *setjmp* and *longjmp* eliminates most dependence on the C stack frame. The remaining machine dependent routines are each about fifteen lines of assembly code on most architectures.

3.1. Excpra — Get Parent's Return Address

The routine *excpra* must return a pointer to the return address for the function that called *exchandle*. For example, in the code:

```
a()
{
        b();
}

b()
{
        exchandle( ... );
}
```

excpra will return a pointer to the word containing the address of the first instruction following the invocation of *b* in *a*. Since *excpra* is itself a subroutine, it must actually return a pointer to its parent's parent's return address (the parent of *excpra* will be *_excvect*, so *excpra*'s parent's parent is the routine that wishes to instantiate the handler.

Most stack frames are designed to facilitate backward searches of this kind. We have only found one C compiler (for the IBM Personal Computer) that presented any unusual difficulties.

3.2. Excdisable — Disable Top Exception Context

_Excdisable is a piece of code (*not* a procedure) that is executed when a procedure that set a handler returns. It can be thought of as an addition to the procedure epilogue. When it starts executing the procedure that set a handler will have already returned, so *_excdisable* is actually running in the context of the procedure that called the procedure setting the handler. For example, in the above illustration *_excdisable* executes in *a*'s context after *b* returns, but before *a* regains control.

_Excdisable needs to call *_excdrop* to drop the exception context corresponding to the procedure context that was just dropped (as a result of *b* returning). *_Excdrop* will return the address in *a* following the invocation of *b*, that is, the address to which *_excdisable* should return.

In some cases it may be easiest to create a new stack frame using a *call* instruction and then modify the return address.

EXAMPLE 12: ▽

A sample *_excdisable* in pseudo-code:

```
_excdisable:
        call    pseudo
pseudo:
        push    return_value
        call    _excdrop
        move    return_value, return_address
        pop     return_value
        ret
```

The first *call* instruction sets up a new stack frame. The *push* instruction saves the return value from the function that is currently returning. The second *call* instruction calls the C code that does the hard work of unwinding the exception stack. It returns the original return address of the function that is returning. The *move* instruction moves this address into the stack frame created by the first *call* instruction. The *pop* restores the previous return value, and the *ret* returns to the original calling function; it does not return to *pseudo* because the return address was replaced by the *move* instruction. Δ

3.3. Experience

So far we have successfully brought up the exception handler on the PDP-11, VAX, 3B20, 68000, and PYRAMID processors with no difficulty other than the typical lack of documentation of the stack frame. Examination of the code produced by the C compiler is usually sufficient to determine the necessary information.

4. ANALYSIS

4.1. Comparison with Other Facilities

Other published exception handlers (e.g., [Goodenough75], [Ada79a], [Mitchell79], [Liskov79]) integrate the exception handler into the host language, that is, the language is extended to include syntax and semantics to support the exception handler. Advantages to this approach are:

- Since exceptions are identifiers rather than strings (and are typically declared) compile-time validity checking is possible.
- Exception scoping can accurately match the scoping of the language, that is, exceptions can be associated with blocks as well as with procedures.
- Exception handlers can access local variables of the routine in which the handler is set. In some cases they can also reference control structures (e.g., **goto** labels and loop **break**'s).
- Exception handlers can specify override return values from a function.

We rejected extending C for several reasons: our portability requirements obviated compile-time exceptions because of the overhead and possible lack of source code; the extension seemed questionable considering that C is noted as a minimalist language; and the benefits of such extensions did not seem to merit the costs.

All the proposals or implementations in the following subsections are integrated into the language and thus have the attributes described above.

It will be seen that the major differences (beyond details of syntax) can be divided into the following areas:

- Do exceptions take arguments?
- Can an exception continue after being raised, or is it forced to back out?

- Are exceptions passed back through the runtime stack automatically, or do they only pass to the immediate caller?

4.1.1. As presented in [Goodenough75]

Goodenough describes an exception handling facility embedded into PL/I. His handlers may be attached ("have a *reach*") to any expression or subroutine call. For example:

(A + B) [OVERFLOW: ...]

will add *A* to *B* and transfer control to the "..." part if an overflow occurs during the addition.

Handlers may have several dispositions:

- The EXIT statement specifies a value to return for the expression that raised the exception. For example:

  IF (A + B > C) [OVERFLOW: EXIT(TRUE);]
  THEN ...

  will cause the test to succeed if an overflow occurs during the addition.

- EXIT without a value causes the *statement* to terminate immediately. For example:

  IF (A + B > C) [OVERFLOW: EXIT;]
  THEN ...
  ELSE ...

  will execute *neither* the THEN nor the ELSE clause if an overflow occurs during the addition.

- RESUME returns control to the point where the exception was raised.

The EXIT construct is roughly equivalent to our backout disposition. RESUME matches our continue disposition.

Goodenough also does not discuss passing parameters other than to note that parameters are an important issue that require further work.

Goodenough types exceptions: ESCAPE exceptions *require* termination of the operation; NOTIFY exceptions *forbid* termination of the operation; and SIGNAL exceptions *permit* termination or continuation. Our Abort severity matches the ESCAPE semantics; all other severities match SIGNAL exceptions. We have no concept of NOTIFY exceptions, although the severity could be used to encode them fairly easily.

Goodenough implements cleanup on backout by raising a special CLEANUP exception.

4.1.2. Ada

The Ada committee took the attitude that exceptions were solely to process error conditions [Ada79b]. As a result, they only support backout (that is, there is no way for an exception to continue). A **return** statement within the handler returns from the block handling the exception. For example:

```
begin
        ...
exception
        when OVERFLOW =>
                return FLOAT'LARGE;
        when others =>
                raise FAILURE;
end;
```

will return the largest possible float value if an overflow occurs during processing of the

block and will raise FAILURE if any other exceptions occur.

Ada exceptions have no arguments.

4.1.3. Mesa

Mesa's *signal* facility [Mitchell79] is especially ambitious: besides providing a way to flag exceptional conditions, Mesa also assists with error recovery.

Unlike our exception handlers, Mesa *catch phrases* may return values. For example, a storage allocation package that ran out of memory could execute:

```
newblock ← signal BlockTooLarge[zone, n];
```

If a new block can be found that will satisfy the request it can be returned using:

```
resume[newblock];
```

primitive and the allocation can continue. Catch phrases may also **retry** the expression raising the signal, **continue** after the expression raising the signal, or ***unwind*** back to the context setting the catch phrase. Unwind may be initiated by **goto**, loop , (similar to **continue** in C), or **exit** (similar to **break**). These are all evaluated in the context that set the catch phrase.

The ability to return values from a handler simplifies recovery. For example, in our scheme a memory allocator allowing recovery would have to execute:

```
if (exchandle("I:GOTMEM", excbackout) == 0)
{
        excraise("E:NOMEM", NULL);
        excraise("A:STILLNOMEM", NULL);
}
```

A handler willing to find memory would catch E:NOMEM, set a global pointer, and raise I:GOTMEM to continue; if E:NOMEM returned the A:STILLNOMEM would abort the allocation request.

Mesa implements our **Abort** severity exceptions using the **error** keyword. **Errors** are otherwise identical to **signals**.

The special signal *Unwind* is reserved to indicate that a context is being unwound. Thus, catching *Unwind* allows cleanup actions.

4.1.4. Clu

Clu [Liskov79] does not allow exceptions to continue, much like Ada. The **signal** statement *immediately* unwinds the procedure context and evaluates the appropriate **except** clause in the context in which it was set. This clause may execute **break** or **continue** statements (which are identical to the same statements in C). The **exit** statement is identical to **signal** except that it executes in the current context (that is, does not unwind procedure contexts at all). A backward search of the stack is *not* performed; any exception to be passed back more than one level must be explicitly reraised.

EXAMPLE 13: ▽

To use signals in a search loop[3]:

[3]We have taken liberties with the Clu syntax in this example to make it more familiar to C programmers.

```
begin
        for i in x do
                if special(i) then exit found end
                end
        i := make_new_one(...)
end except when found: end
```

This example searches through the elements of the array *x* for a "special" element. The loop terminates by raising the *found* exception when *special(x)* is satisfied. The **except** condition does nothing in this case; it is used only to prematurely terminate the loop. If this element is not found a new element is created. In either case *i* is available for use when this code fragment completes. Δ

Clu has no builtin facility to handle cleanup processing. This is not strictly necessary since exceptions are never implicitly passed through a context, that is, **except** clauses must catch every exception explicitly.

4.1.5. PL/I

"Exceptions" in PL/I are not true exceptions as we have defined them here. They do not have backout[4] and cannot be nested. There are a small fixed number of language-defined exceptions. Each handler may be a routine to gain control on the event in question or a default action. The routines do not have access to local variables, do not have parameters, and cannot specify return values from the raising procedure.

PL/I exceptions most closely match our default handlers.

4.2. Semantic Gotchas

4.2.1. Interaction with resources

Non-local **goto**'s of any form, including exceptions, will interact with any resources allocated from a global pool such as file descriptors or memory. For example, in the code segment:

```
f()
{
        char *x;
        x = malloc( ... );
        excraise("E:X", NULL);
}
```

the memory referenced by *x* must be freed *if and only if* the handler associated with "E:X" decides to back out. This is processed using cleanup handlers:

```
f()
{
        char *x;
        x = malloc( ... );
        exccleanup(free, x);
        excraise("E:X", NULL);
}
```

Integrating the exception handler into the host language can help for some of these problems. However, this also requires that the resources in question also be

[4]Although this can be simulated with non-local **goto**'s; contexts being unwound do not have the option of gaining control during backout to perform cleanup actions.

integrated into the language. This is contrary to the current trend in algorithmic languages. Cleanup handlers are still required in any case to free "simulated resources" – for example, aborting transactions in a database system or cleanly terminating a network protocol.

Integration does allow direct reference to local variables. For example, the following syntax could be envisioned:

```
f()
{
        char *x;
        x = malloc( ... );
        excraise("E:X", NULL);
}
cleanup
{
        free(x);
}
```

The **cleanup** clause would inherit all the local variables of *f*.

4.2.2. Static arguments

We encountered an ugly pragmatic problem fairly early: the "call by address" semantics of string parameters could cause apparently incorrect parameters. For example, the call:

```
excraise("E:X", errstring(errno), NULL);
```

appears innocuous, but can create problems if *errstring* can return a pointer to a static buffer, since an error during processing of the exception can change the value of the buffer.

When we discovered this problem enough code had been written that we simply decided to copy the arguments. There appears to be no sound theoretical basis for this decision.

4.2.3. Deeper nesting structures

There is no way to regain control during block exit due to C's static processing of blocks. Complete simulation of C scoping is thereby impossible. For example:

```
f()
{
        int i;
        if ( ... )
        {
                int i;
                exchandle("W:X", excignore, 0);
                excraise("W:X", NULL);
        }
        excraise("W:X", NULL);
}
```

Ideally the second *excraise* call would not be caught by the handler *excignore*, much as a reference to *i* inside the block references a different *i* than outside the block. However, this would require that the exception package seize control when the block exits.

This could of course be handled if exceptions were integrated into the language. Most other facilities declare the possible exceptions and handlers much like variables. For example, one can imagine the previous example having the following syntax if

exceptions were integrated into C[5]:

```
f()
{
        if (...)
        {
                int i;
                raise W:X;
        } when W:X do excignore();
        raise W:X;
}
```

4.3. Strings as Exception Identifiers

Our original implementation of exceptions used integers as exception identifiers instead of strings. We dropped this for four reasons:

- In order to allow user-defined exceptions we had to have "allocate exception" primitives. This created a new resource that was difficult to manage.
- An integer namespace is difficult to correlate externally. We also maintain a database of messages associated with each exception. This database gives us a message to print if no handler catches the exception and also serves to document the semantics of the exceptions. Using strings simplified development of this database.
- Management of many exceptions (in our current system we have over 600 distinct exceptions) is much easier with meaningful names.
- It is easier to specify arbitrary patterns on strings. For example, we can specify:

```
exchandle("*SYNTAX*", ...)
```

to catch all syntax errors generated by all modules and parsers.

Instead of using strings, we could have defined a new data type. For example:

```
f()
{
        extern EXCEPTION e;
        exchandle(&e, ... );
}
```

This still would have had many of the problems seen with integers.

4.4. Strings as Exception Parameters

For our purposes using strings as exception parameters has been an acceptable compromise, although at times it has proven inconvenient. In some environments it might be reasonable to allow arbitrary types (much as *printf*(3) allows arbitrary types), although this obviates copying string parameters, thereby creating the problems noted above.

4.5. Keeping Stacks in Sync

We were originally concerned that the exception stack might become out of sync with the language runtime stack. In practice this has not been a problem.

Extra tests could be added to test for this error condition if it became appropriate. For example, if we were willing to make some other assumptions about the layout of the runtime stack we might store the frame pointer register in the exception context to help validate the exception stack.

---

[5]This example includes a number of syntactic idiosyncrasies which would have to be resolved in a serious proposal.

5. PERFORMANCE

Five benchmarks were used to test the performance of the exception handler. Benchmarks were run on a three megabyte VAX/750 running 4.2BSD in single user mode. Each program was first run with a small test size to page in the program immediately followed by another run which executed a loop 10,000 times. User and System times were determined via *getrusage*(2). Startup time was not included in the usage times. See section 5.3 for a discussion of the test results and the Appendix for listings of the benchmarks. All times are given in milliseconds.

5.1. Longjmp and Setjmp

These routines were benchmarked to determine their contributions to excraise and exchandle respectively. See the "Normal" times in Table 2. Times for *longjmp* are subtracted from the backout times while the *setjmp* times are subtracted from context drop times.

	Set Jump	Long Jump	Sum
Total	4006.67	2875.00	6881.67
Per Call	0.40	0.29	0.69
User	0.09	0.07	0.16
System	0.31	0.22	0.53

Table 1. Setjmp/Longjmp Benchmark Results (times in msec)

5.2. Ignore, Backout, and Drop

The *ignore* benchmark sets one handler for the exception code "E:HI" and raises this exception 10,000 times. When the exception is raised the exception stack is searched for the handler, and this handler is called. The handler (*excignore*) returns 0 to continue execution from where the exception was raised. The time spent to set the handler is not included in the numbers.

Backout is similar to the ignore test except that the handler *excbackout* returns 1 causing control to pass to where the exception handler was set rather than to continue at the raise call. *Longjmp* is called to pass control to the point at which the handler was set. Ignore does a simple return from *excraise*. The *Normal* time excludes the *longjmp* time.

The *drop* benchmark calls a routine *f* 10,000 times who sets from one to twenty unique handlers. Entering *f* allocates a new context on the exception stack when *exchandle* is called. An exception vector is then allocated for each exception code and linked to the front of the vector list for *f's* context. When *f* returns the vectors and the context are freed. The *Normal* times exclude the cost of *setjmp*.

	Ignore	Backout		Drop 1		Drop 20	
	Total	Total	Normal	Total	Normal	Total	Normal
	12735.00	17160.00	14295.00	18710.00	14703.33	491410.00	411276.60
Per Call	1.27	1.72	1.43	1.87	1.47	49.14	41.14
User	1.26	1.43	1.36	1.50	1.41	41.88	36.68
System	0.01	0.29	0.07	0.37	0.06	7.66	1.46

Table 2. Exception Benchmark Results (times in msec)

5.3. Discussion

Times for *setjmp* and *longjmp* were gathered to assess how much time they contributed. The *backout* and *drop* benchmarks indicate that most of the system time was in *longjmp* and *setjmp* respectively.

A *gprof*(1) analysis was performed to determine where time was spent in the exception facility.

When an exception is raised memory is allocated to save a copy of the parameters. This copy is required because the parameters may point to static memory that can be overwritten during exception processing. The time attributed to allocating and freeing memory is approximately 30% to 40% of the time spent within the *excraise* call.

The time attributed to building and freeing a context measured in the "noise" (zero seconds).

In the second drop case (twenty handlers[6]) most time was found to be checking previous handlers' exception codes to see if they matched the exception code of the exception being set. Matching vectors are freed since the previous handler is no longer reachable. Most of the remaining time is in the memory routines.

6. SUMMARY AND CONCLUSIONS

We have described a design and implementation of an exception handler for the C language that does not require any language changes and which uses only minimal knowledge of the structure of the runtime stack. We have used this facility for over a year and have found it a useful software tool.

Failure to integrate exceptions into the language results in a few problems:

- Exception handlers cannot run in the context of the procedure that set the handler, and hence cannot have access to local variables. This makes some operations more difficult to implement.
- Parameters to exceptions are constrained to be character strings. This could be relaxed without integration, but all type checking would be forfeited.
- Names of exceptions are completely uncontrolled.

Despite these problems, we feel this implementation compares favorably to many of the comparable facilities in other languages.

ACKNOWLEDGEMENTS

We owe the fundamental observation that an exception handler could be implemented in C without language changes to Kurt Shoens, who graciously permitted us to steal his idea. Peter Ford used the handler in bold new ways, demonstrating many important improvements to us.

REFERENCES

[Ada79a] "Ada Reference Manual." Published as *ACM SIGPLAN Notices 14*, 6, Part A. June 1979.

[Ada79b] "Rationale for the Design of the ADA Programming Language." Published as *ACM SIGPLAN Notices 14*, 6, Part B. June 1979.

[Darwin85] Darwin, I., and Collyer, G., "Can't Happen -or- /* NOTREACHED */ -or- Real Programs Dump Core." In *Proc. Winter 1985 USENIX Conference*. Dallas, Texas. January 1985.

[6]Twenty handlers were chosen as an extreme case. This is atypical in most applications of the exception handler.

[Dijkstra68] Dijkstra, E. W., "Go To Statement Considered Harmful." In *Comm. ACM 11*, 3. March 1968.

[Goodenough75] Goodenough, J. B., "Exception Handling: Issues and a Proposed Notation." In *Comm. ACM 18*, 12. December 1975.

[Knuth74] Knuth, D. E., "Structured Programming with Gotos." In *Computing Surveys 6*, 4. December 1974.

[Liskov79] Liskov, B., *et al*, *Clu Reference Manual.* MIT/LCS/TR-225. Laboratory for Computer Science, Massachusetts Institute of Technology. Cambridge, Massachusetts. October 1979.

[Mitchell79] Mitchell, J. G., Maybury, W., and Sweet, R., *Mesa Language Manual* Version 5.0. CSL-79-3. Xerox Palo Alto Research Center, Systems Development Department. April 1979.

A. Appendix

The following are example code segments used in running the benchmarks.

A.1. Backout Handlers

```
test()
{
        if (exchandle("E:HI", excbackout) == 0)
        {
                /* get time after the handler (once) */
                (void) getrusage(RUSAGE_SELF, &r1);
        }
        if (++i ≥ TestSize)
        {
                (void) getrusage(RUSAGE_SELF, &r2);
                printstats(r1, r2);
                exit(RS_NORM);
        }
        /* backout to where handler is set */
        excraise("E:HI", CHARNULL);
}
```

A.2. Dropping Handlers

```
test()
{
        int i = 0;
        (void) getrusage(RUSAGE_SELF, &r1);
        while (i++ < TestSize)
                f();
        getrusage(RUSAGE_SELF, &r2);
        printstats(r1, r2);
        exit(RS_NORM);
}

f()
{
        int i;
        char **e;
```

```
        /* set NumHand uniquely named handlers - case 1 = 1, case 2 = 20 */
        for (i = 0, e = Elist; i < NumHand && *e ≠ CHARNULL; i++, e++)
                exchandle(*e, f);
        return;
}
```

A.3. Ignoring An Exception

```
test()
{
        if (exchandle("E:HI", excignore) == 0)
        {
                /* get time after the handler (once) */
                (void) getrusage(RUSAGE_SELF, &r1);
        }
        for (i = 0; i < TestSize; i++)
                excraise("E:HI", CHARNULL);
        (void) getrusage(RUSAGE_SELF, &r2);
        printstats(r1, r2);
}
```

A.4. Set Jump

```
test()
{
        (void) getrusage(RUSAGE_SELF, &r1);

        for (i = 0; i <= TestSize; i++)
                setjmp(env);

        (void) getrusage(RUSAGE_SELF, &r2);
        printstats(r1, r2);
}
```

A.5. Long Jump

```
test()
{
        if (setjmp(env) == 0)
        {
                /* get time setjmp (once) */
                (void) getrusage(RUSAGE_SELF, &r1);
        }
        if (++i ≥ TestSize)
        {
                (void) getrusage(RUSAGE_SELF, &r2);
                printstats(r1, r2);
                exit(0);
        }
        longjmp(env, 1);
}
```

A.6. Printstats

```
/*
** Print the results.
*/
printstats(r1, r2)
        struct rusage *r1, *r2;
{
        long usec = 0;
        long ssec = 0;
        long uusec = 0;
        long susec = 0;
        float systime;
        float usertime;

        /* determine total user and system time */
        usec  += (r2.ru_utime.tv_sec  - r1.ru_utime.tv_sec);
        uusec += (r2.ru_utime.tv_usec - r1.ru_utime.tv_usec);
        ssec  += (r2.ru_stime.tv_sec  - r1.ru_stime.tv_sec);
        susec += (r2.ru_stime.tv_usec - r1.ru_stime.tv_usec);

        /* convert seconds back to micro-seconds */
        usertime = usec * 1000000;
        systime  = ssec * 1000000;
        usertime += uusec;
        systime  += susec;

        tprintf("TOTAL:%.2fmsec        %.2fmsec\n",
                        usertime/1000, systime/1000);'
        size *= 1000;
        tprintf("AVG  : %.2fmsec/test  %.2fmsec/test\n",
                        usertime/size, systime/size);
}
```

si—An Interpreter for the C Language

Alan R. Feuer

55 Wheeler Street
Catalytix Corporation
Cambridge, Mass 02138

(617) 497-2160
cataly!arf

ABSTRACT

An interpreter for the C language has been developed that offers several advantages during program development over the traditional C compiler/linker/debugger: faster turnaround for modifications, finer control over program execution, better debugging facilities, and more complete run-time error checking. The interpreter is named **si**, the Safe C Interpreter.

This paper begins by describing those features of **si** that distinguish it from the traditional C development environment. Next it contains a sample session that illustrates the interaction between a programmer and the Interpreter. It concludes with a discussion of a few key design decisions made during the development of **si**.

INTRODUCTION

The advantages of highly interactive program development environments have been described widely [GOLD83, KAY69, REIS84, SAND78, TEIT81]. All promise more productive software development at the expense of more computer cycles; a good tradeoff, these days.

This paper describes **si**, an interpreter for the C language that, while not a full-fledged programming environment, shares many attributes with the more ambitious projects. Rather than supplanting its host, **si** has been designed to blend into its operating environment.[1] Its major advance is that it provides for C the fast turnaround, good debugging tools, fine control over execution, and extensive error checking of an integrated programming system without sacrificing the utility of existing program development tools.

MAJOR FEATURES

si was designed to be a programming tool for application programmers.[2] It handles multi-file, multi-directory programs, it works with any text editor or special preprocessor, it can access compiled code, and it understands the full C language.

The first attribute of the Interpreter that attracts people used to compiling C is the short cycle between editing a program and running it. By removing from the edit-run cycle compiling, assembling, and link editing, changing a program becomes a low-penalty operation.

---

1. The Interpreter is running under UNIX, VMS, MS/DOS, and on the Macintosh. However, not all implementations have all features.
2. A lean version of the Interpreter is being packaged with a textbook for use in teaching C.

Link editing is done at run time as needed. One consequence is that pieces of a program can be run in isolation. As the functions of a program are developed they can be tested without the need to generate temporary driver routines.

Run-time errors, such as using a variable before it is defined or indexing/pointing beyond the end of an array (even for dynamic arrays), are trapped. Compiled programs that work by taking advantage of system-dependent circumstances, e.g., a NULL at location zero or physical adjacency of separately defined variables, generate run-time errors while being interpreted.

Parameters to functions are also checked at run time. The number and types of the parameters in a function call are checked against the function definition. For the formatted I/O routines of the **printf/scanf** family, the conversion specifications in the control strings are checked against the types of the remaining parameters passed. For routines in the standard libraries, the value of the passed parameters are checked for sanity. For example, in a call to **fseek**, if the origin is from the start of the file then the offset is checked to make sure it is positive.

There are several ways to stop a program in the Interpreter during its execution: A run-time error will halt the program immediately; an interrupt from the keyboard will stop the program at the next semicolon in the source text; a program can be single-stepped, stopping at each semicolon; assertions inserted into the source code will halt the program if they fail; explicit breakpoints can be set either in the code or from the command line.

When a program is stopped, the Interpreter accepts C expressions from the standard input. Each expression entered is evaluated in the context of the stopped program just as though it were inserted at the current point of execution.

A program can be traced while it executes. Function call/return, statement execution, and expression evaluation can each be traced separately or in combination.

SAMPLE SESSION

The sample session that follows shows the Interpreter running on the UNIX system. In the displays, user input appears in **bold** font and comments are printed in *italics*.

Basic Interaction

Interaction with the Interpreter is modeled after program development on the UNIX system. The Interpreter operates in two modes: *command* (the Shell) and *execution* (C). On invocation, the Interpreter is in command mode:

```
$ si      invoke si
si.$
```

Command mode is used to establish the environment for running a program. In the file system, programs are kept in files; within the Interpreter they are packaged in modules. A *module* is the internal representation of a file. When a file is read into the Interpreter a corresponding module is created:

```
si.$ r echo.c    read a file and create a module
loading "./echo.c"
si.$ p    print the current module
echo (in file ./echo.c)
 1   main(argc,argv)
 2   int argc;
 3   char *argv[ ];
 4   {
 5      int i;
 6
 7   STOP( );
 8      for( i = 1; i<argc; + + i )
 9            printf("%s%c", argv[i], i<argc—1?' ':'\n');
 10  }
si.$
```

Programs can be executed in the Interpreter using syntax modeled after the Shell. Shell quoting, environment variables, pattern matching, and I/O redirection are all supported:

```
si.$ echo \'Twas brillig "at $HOME"
STOP on line 7 in function "main" of module "echo"
.
```

Execution mode is entered whenever a program is stopped. One way to stop a program is to call the built-in routine **STOP**, as on line seven of **echo**. In execution mode the Interpreter accepts C expressions from the standard input, evaluates them, and prints the resulting value and type. Any expression can be entered, including calls to functions and references to active variables:

```
. argc
= (int) 3
. argv[0]
= (char *) 643492 (0x9D1A4) "echo"
. PR(argv)
argv (char **)
        [0]      = 643492 (0x9D1A4) "echo"
        [1]      = 643504 (0x9D1B0) "'Twas"
        [2]      = 643516 (0x9D1BC) "brillig"
        [3]      = 643528 (0x9D1C8) "at /usr/arf"
        [4]      = <NULL>
PR on line 7 in function "main" of module "echo"
= (void) <UNDEFINED>
```

The built-in function **PR** expands the first level of structure for aggregate data. It does not return an explicit value, so the result of the call is **<UNDEFINED>**.

Variables can be modified using standard C syntax:

```
. argv[1] = "'Twasn't"
= (char *) (644044,644053) 644044 (0x9D3CC) "'Twasn't"
```

The pair of numbers printed in the parentheses above are the bounds of the pointer **argv[1]**.

Execution of the stopped program continues following an end-of-file (**<eof>**):

```
. <eof>
'Twasn't brillig at /usr/arf
si.$
```

To change a module, the Interpreter invokes a user-specified text editor on the file associated with the module. In this example I will use the **ed** editor because it matches the paper medium best:

```
si.$ e  edit the current module
134
7d      delete the call to STOP
w
126
q
loading "./echo.c"
si.$
```

After editing, the Interpreter rereads those files that have been changed since the last time they were read. **echo** now reflects the change, i.e., it no longer stops on line seven:

```
si.$ echo my text editor is called $editor
my text editor is called /bin/ed
si.$
```

Watching a Program Execute

A program can be traced on any of three levels: expression, statement, and function. Each level is controlled by the setting of a flag:

```
si.$ +te        enable tracing of expressions
—d —l —ss +te —tf —ts
```

In an expression trace, the operands to an operator appear before the operator:

```
si.$ x  enter execution mode
. 3.4 + 5*6
        3.4 = (double) 3.4
              5 = (int) 5
              6 = (int) 6
        5*6 = (int) 30
3.4+5*6 = (double) 33.4
= (double) 33.4
. 1 || 0
        1 = (int) 1
1||0 = (int) 1
= (int) 1
. 1 && 0
        1 = (int) 1
        0 = (int) 0
1&&0 = (int) 0
= (int) 0
```

The levels of tracing can be mixed in any combination. Here is a statement and function trace of a recursive power function:

```
si.$ r power.c
loading "./power.c"
si.$ p    power is now the current module
power (in file ./power.c)
1   power(x,n) int x, n; { /* return x to the nth power */
2       if( n>0 ) return( x*power(x,n—1) );
3       else return(1);
4   }
si.$ —te,+tf,ts          expression trace off, function and statement trace on
—d  —l  —ss  —te  +tf  +ts

. power(3,2)
        power(3,2)
power:  n= (int) 2
        x= (int) 3
if( n>0 )
return( x*power(x,n—1) )
power:  n= (int) 1
        x= (int) 3
if( n>0 )
return( x*power(x,n—1) )
power:  n= (int) 0
        x= (int) 3
if( n>0 )
return( 1 )
power = (int) 1
power = (int) 3
power = (int) 9
= (int) 9
```

Detecting Run-Time Errors

Run-time errors such as indexing beyond the end of an array, marching a pointer off the end of an object, passing the wrong arguments to a function, or using a variable before it is defined are caught by the Interpreter. When a run-time error is detected, the program is stopped and a new expression evaluator, in Lisp-like fashion, is spawned:

```
. $      $ is a generic variable
= (void) ⟨UNDEFINED⟩        initially $ is undefined
. $ = malloc(—10)
bad argument to "malloc"
        int arg should be >= 0
STOP
. .
```

Entering end-of-file returns to the previous evaluator:

```
. . ⟨eof⟩
= (void) ⟨UNDEFINED⟩
. $ = malloc(10)          $ takes on the type of the value assigned to it
= (char *) (403024,403034) 403024 (0x62650) ""
. PR($)
$ (char *)
        [0]     = '^@' (00)
        [1]     = '^@' (00)
        [2]     = '^@' (00)
        [3]     = '^@' (00)
        [4]     = '^@' (00)
        [5]     = '^@' (00)
        [6]     = '^@' (00)
        [7]     = '^@' (00)
        [8]     = '^@' (00)
        [9]     = '^@' (00)
PR
= (void) ⟨UNDEFINED⟩
```

The generic variable **$** is now a pointer to a 10-element character array. A pointer to an object is bounded by the object, hence stray references can be detected:

```
. $[9]
= (char) '^@' (00)
. $[10]
pointer/index out of range
STOP
. . *($ + 10)
pointer/index out of range
STOP
. . ⟨eof⟩
= (void) ⟨UNDEFINED⟩
```

The result of an erroneous operation is always **⟨UNDEFINED⟩**. **⟨UNDEFINED⟩** is also the return value of a function that returns no value and the initial value of uninitialized automatic variables.

Linking to Compiled Functions

Compiled modules are manipulated similarly to interpreted ones. First the compiled file is read and a module is created:

```
si.$ !cc —c power.c     create a compiled file
si.$ r power.o          read the compiled file
"power" already loaded, type o to overwrite: o
loading "./power.o"
```

Since the interpreted version of **power** was still loaded, the Interpreter asked for confirmation to replace it with the compiled one. When a compiled module is loaded, user-specifiable libraries are searched to satisfy unresolved references. Thus the compiled module may contain functions that were not in the source:

```
si.$ f power    list the functions in power
power (in file ./power.o) compiled
        lmul( ) : (int)
        mul( ) : (int)
        power( ) : (int)
```

The compiled version of **power** can be executed just like the interpreted one:

```
si.$ x
. power(3,2)
= (int) 9
```

DISCUSSION

A few fundamental design decisions have been responsible for much of the nature of **si**.

Tools versus Environments

Innovation in programming tools continues [HEND84]. The current vogue is to weave these tools into an integrated environment to achieve conceptual cleanliness and added efficiency. Our goal with **si** is not so ambitious.

Without joining into the tools vs. environment fray, it is clear that **si** is a tool, not an environment; it can be used alongside a programmer's other tools. Making the decision to build a tool made other design decisions easy:

> *Should* **si** *contain its own editor?* No. There are real advantages to building an editor into an interpreter, such as immediate syntax checking and faster context switch between editing and execution. Nevertheless, editors live in our fingers and not our heads; everyone has their favorite editor. And if you like a particular syntax-directed editor, **si** won't stop you from using it.
>
> *Should* **si** *contain its own preprocessor?* No. C is evolving in a direction that encourages embedding **cpp** into the translator. On some systems, embedding **cpp** results in a nice performance gain. But the existence of special preprocessors has shown that having **cpp** as a separate program makes C more malleable. For instance, C++ [STRO83] was first implemented as a special preprocessor to C. The Safe C Runtime Analyzer [FEUE85] is a source-to-source transformer that runs after **cpp** and before the first pass of the C compiler.
>
> *Should* **si** *control the screen like a visual editor?* No. The Interpreter would be easier to use if the different kinds of output were directed to different places on the screen. Some development environments divide the screen into different windows, one showing the currently executing text, another showing the values of interesting variables, a third showing the output of the running program, and another used for commands to the system.
>
> In the implementation of **si**, we have tried to separate the user-interface from the Interpreter engine. To assist in building a clean interface, the different types of output from the Interpreter are sent to distinct streams. If the underlying system has the flexibility to attach those streams to different windows, then **si** can be given a multi-window front-end. The interface built into **si** is easily transportable across systems and does not compete with the program being interpreted.

One mode versus many

A central feature of some programming environments is that they operate in one mode; expressions in a universal language are always acceptable [DELI84, GOLD83, SAND78]. The traditional C development environment on the UNIX system embodies several modes and at least two languages: the Shell and C. With the Interpreter we had the opportunity to drop one of the languages, presumably the Shell, but chose instead to stick with the UNIX model.

Our feeling is that C does not make a wonderful command language. Since a command language is interactive, we prefer short commands with little extraneous punctuation. Typing

```
r("power.c")
```

instead of

```
r power.c
```

does not seem worth the conceptual purity.

In addition, the Shell has shown us the utility of I/O redirection and pattern matching in a command language. Since these are not features of C, using C as a command language would force us to give up the convenient and familiar Shell syntax for something bulkier.

On the other hand, C seems like the natural choice for the language to explore a C program. With this decision we leave ourselves slightly vulnerable because we have just argued that C does not make a good command language. Browsing surely requires a command language of some sort.

Our experience with debuggers that have opted for an interactive language instead of C convinces us that, in this context, it is worth the extra keystrokes to have the versatility of C. In providing all of the peeking and poking commands, debugging languages invariably become too complex. C has all the requisite power plus the advantage of familiarity.

Translation-time versus run-time checking

Standard practice when designing a compiler is to do as much checking at translation time as is possible, thus saving run-time overhead. We observe that during program development, more time is spent editing, compiling, assembling, and linking than running. By delaying some checks until run time, translation time is reduced.

In addition, during program development it is advantageous to have complete checking. For some important classes of errors, run-time checking is required for completeness. Since the Interpreter links on demand allowing pieces of a program to be executed in isolation, the referent for a function is not known (and may not exist!) until run time. Thus type checking of function calls across modules cannot be done at translation time. The use of a variable before it has been assigned a value, indexing outside the bounds of an array, and indirection through a stray or dangling pointer could each be detected in some cases at translation time. Complete checking for these errors, however, can only be done at run time.

CONCLUSION

si is a young program, it has been available outside of Catalytix for just five months. si continues to evolve based on feedback from its users. The most active areas of evolution deal with the user-interface and the performance of interpreted programs.

Within Catalytix, the Interpreter has become the translator of choice for developing new programs. It offers superior error checking and good tools for debugging at a tolerable cost in run-time performance. Our compilers have been relegated to what they do best, translating already debugged programs for fast execution.

REFERENCES

DELI84 Delisle, Norman M., David E. Menicosy, and Mayer D. Schwartz, "Viewing a Programming Environment as a Single Tool," *Proc. ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments*, Pittsburgh, PA, April 1984.

FEUE85 Feuer, Alan R., "Introduction to the Safe C Runtime Analyzer," Catalytix Corp. Technical Report, January 1985.

GOLD83 Goldberg, A. J. and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.

HEND84 Henderson, Peter (editor), *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* Pittsburgh, PA, April 1984.

KAY69 Kay, Alan Curtis, *The Reactive Engine*, Ph.D. Thesis, Department of Computer Science, University of Utah, 1969.

REIS84 Reiss, Steven P., "Graphical Program Development with PECAN Program Development Systems," *Proc. ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments*, Pittsburgh, PA, April 1984.

SAND78 Sandewall, Erik, "Programming in an Interactive Environment: the 'Lisp' Experience," *Computing Surveys*, vol 10, no 1, March 1978.

STRO83 Stroustrup, B., "Adding Classes to C: An Exercise in Language Evolution," *Software—Practice and Experience*, vol 13, pp 139-61, 1983.

TEIT81 Teitelbaum, T. and T. Reps, "The Cornell program synthesizer: a syntax-directed programming environment," *Communications of the ACM*, vol 24, no 9, Sept. 1981.

An Extensible I/O Facility for C++

Bjarne Stroustrup

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill NJ 07974

ABSTRACT

This paper describes the classes `ostream` and `istream` designed to replace the `printf/scanf` family of input and output functions in C++. The operator `<<` is overloaded for ostreams to provide an single type-secure paradigm for output of both built-in and user-defined types. This paradigm typically yields output statements as short as or shorter than `printf()`. The operator `>>` handles input in a similar fashion. Conditions like reaching the end of a file or the corruption of a stream are handled by associating a state with each stream, rather than by returning "illegal values" like `EOF`. The C++ facilities used to implement stream i/o are briefly explained. These include classes providing data hiding, constructors providing initialization, destructors providing cleanup (in particular, flushing of buffers), operator overloading, and virtual functions providing uniform use of buffers with different strategies for handling underflow and overflow.

Introduction

Except for minor details C++ is a superset of the C programming language. In addition to the features of C, C++ provides Simula67-like classes, operator overloading, and a host of minor improvements. The parts of the C++ language that are used to specify the stream I/O facility are briefly explained. References 6-8 describe C++ in the detail necessary for writing programs in it.

The `printf()` family of functions provides simple, flexible, and terse formatted output[5] for C programs (and therefore also for C++ programs). However, uses of `printf()` cannot in general be type checked, and there is no convenient way for a user to deal with user-defined types in the same way as built-in types. Consider:

```
printf(stderr,"x = %s\n",x);
```

This is of course an error under all circumstances since only `fprintf()` and not `printf()` takes a `FILE*` like `stderr` as an argument. This problem is trivially handled in C++ where `<stdio.h>` declares

```
extern int fprintf(FILE*, char* ...);
extern int printf(char* ...);
```

thus catching that error at compile time. The ellipsis indicates that any number of arguments of any type may follow the initial arguments. However, had `x` been an `int` (rather than the `char*` expected by the `%s` in the format string) no error would have been detected until the program started printing garbage. Furthermore, had `x` been a user-defined type like `complex` there would have been no way of specifying the output format of `x` in the convenient way used for types "known to `printf()`" (for example, `%s` and `%d`). The programmer would typically have defined a separate function for printing complex numbers and then written something like this:

```
fprintf(stderr,"x = ");
put_complex(stderr,x);
fprintf(stderr,"\n");
```

This is inelegant, and would be a major annoyance in C++ programs where a non-trivial program typically uses several user-defined types for the manipulation of entities that are interesting/critical to an application.

Type-security and uniform treatment can be achieved by using a single overloaded function name for a set of output functions. For example:

```
put(stderr,"x = ");
put(stderr,x);
put(stderr,"\n");
```

The type of the argument determines which "put function" will be invoked for each argument. However, this is too verbose. The C++ solution using an output stream for which << has been defined as a "put to" operator looks like this:

```
cerr << "x = " << x << "\n";
```

where `cerr` is the standard error output stream (equivalent to the C `stderr`). So, if `x` is an `int` with the value `123`, this statement would print

```
x = 123
```

followed by a newline onto the standard error output stream.

This style can be used as long as `x` is of a type for which operator << is defined, and a user can trivially define operator << for a new type. So, if `x` is of the user-defined type `complex` with the value `(1,2.4)`, the statement above will print

```
x = (1,2.4)
```

on `cerr`.

The stream I/O facility is implemented exclusively using language features available to every C++ programmer. Like C, C++ does not have any I/O facilities built into the language. The stream I/O facility is provided in a library and contain no "extra-linguistic magic".

Output of built-in types

For output the class `ostream` is defined. The operator << ("put to") is defined to handle output of the built-in types:

```
class ostream {
        // ...
public:
        ostream& operator<<(char*);     // write
        ostream& operator<<(long);      // beware: << 'a' writes 97
        ostream& operator<<(double);

        ostream& put(char);             // put('a') writes a
        ostream& flush();

        // ...
                ostream(streambuf* s);          // bind to stream buffer
                ostream(int fd);                // bind for file
                ostream(int size, char* p);     // bind to vector
                ~ostream();
};
```

This class declaration defines the new type `ostream`. A `class` declaration is very much like a C `struct` declaration, except that a C++ class can have function members. Furthermore, a member of a class appearing before the `public:` label can only be used by the functions mentioned in the class declaration and is inaccessible to all other functions in the program. The internal representation of an `ostream` has been rendered inaccessible to a user in this way, and since it is not particularly interesting it has been omitted to simplify the discussion. The comment

```
// ...
```

is used to indicate this. In C++ `//` starts a comment that terminates at the end of line. Traditional C `/* */` comments can also be used.

The operations `<<`, `put()` and `flush()` may be applied to an ostream. For example:

```
cerr << "x = ";
```

where `cerr` is an `ostream` will be interpreted as

```
cerr.operator<<("x = ");
```

An `operator<<` function returns a reference to the `ostream` it was called for so that another ostream can be applied to it. For example:

```
cerr << "x = " << x;
```

where `x` is an `int`, will be interpreted as

```
(cerr.operator<<("x = ")).operator<<(x);
```

In particular, this implies that when several items are printed by a single output statement they will be printed in the expected order: left to right. Since `"x = "` is a string the first `operator<<` function will be chosen to write it, and similarly the appropriate `operator<<` function will be chosen for `x` depending on `x`'s type. Since `x` was an `int` it will be implicitly converted to a `long` (as if in an assignment) and passed to the second `operator<<` function. Floating point numbers are handled by the third `operator<<` function.

This facility for overloading function names and operators and then choosing the correct version to use for a particular call based on the types of the arguments is general in C++ and has nothing particular to do with I/O. Overloading enables the programmer to reduce the number of function names needed in a program by allowing several functions performing similar operations on objects of different types to share a name. In this particular case we avoid the `printf()`, `fprintf()`, and `sprintf()` name proliferation. The implicit type conversions reduce the number of functions needed. For example, there is a single function for handling the integral types: `char`, `short`, `int`, and `long`. There is no facility for printing `unsigned` values in a different way from signed values since the facility for resolving calls to overloaded functions in C++ cannot distinguish signed and unsigned types. Separate functions can, when needed, be used to handle such cases.

Some design details

It was necessary to define an output operator to avoid the verbosity that would have resulted from using an output function. But why `<<`? In C++, it is not possible to define a new lexical token, so one could not simply invent a new operator†.

The assignment operator was a candidate for both input and output, but it binds the wrong way. That is, `cout=a=b` would be interpreted as `cout=(a=b)`, and most people seemed to prefer the input operator to be different from the output operator.

† Part of the reason for this "restriction" is that most "obvious" choices of new operators would create ambiguities and/or render legal C++ programs illegal. Consider, for example, these "possible operators": `->`, `**`, `<-`, and `//`.

The operators < and > were tried, but the meanings "less than" and "greater than" were so firmly implanted in people's minds that the new I/O statements were for all practical purposes unreadable (this does not appear to be the case for << and >>). Apart from that, '<' is just above ',' on most keyboards and people were writing expressions like this:

```
cout << x , y , z;
```

It is not easy to give good error messages for this.

Another problem was that there are no character valued expressions in C++ (exactly like in C). In particular, '\n' is an integer (with the value 10 when the ASCII character set is used), so that

```
cout << "x = " << x << '\n';
```

writes the number 10 after x and not the expected newline. This and similar problems can be alleviated by defining a few macros

```
#define sp << " "
#define ht << "\t"
#define nl << "\n"
```

The example can now be written like this:

```
cout << "x = " << x nl;
```

Using non-syntactic macros is considered bad style in some quarters, but I like these (despite disliking macros in general).

Consider also these examples

```
cout << x << " " << y << " " << z << "\n";
cout << "x = " << x << ", y = " << y << "\n";
```

Most people find them hard to read because of the high number of quotes and because the output operator is visually too imposing. The macros above plus a bit of indentation can help here:

```
cout << x sp << y sp << z nl;
cout << "x = " << x
     << ", y = " << y nl;
```

Initialization and closing of output streams

There are two standard output streams cout and cerr corresponding to stdout and stderr. Naturally, the implementation involves a buffer that is occasionally flushed onto the associated output device. Flushing can be done explicitly like this:

```
cout.flush();
```

This is not required; the buffer is internal and hidden and will be flushed automatically when appropriate.

Two related questions comes to mind: How did a stream like cout get initialized, and how does it get flushed when the program terminates? Consider this program

```
#include <stream.h>

main()
{
        cout << "Hello, world";
}
```

The include directive ensures that the declarations needed to use an ostream are available. When class ostream was declared it was provided with constructors and a destructor. A constructor is a function that must be called whenever an object of its type is created. It is distinguished by the

compiler by having the same name as its class. In particular, for an ostream the constructors

```
ostream(streambuf* s);          // bind to stream buffer
ostream(int fd);                // bind for file
ostream(int size, char* p);     // bind to vector
```

were declared so one of them must be called when an ostream is created. Which one to call will, as usual, be determined by the type of the arguments. The standard output streams are declared like this (using file bufferes as described below):

```
char cout_buf[BUFSIZE];
filebuf cout_file = filebuf(1,cout_buf,BUFSIZE); // UNIX output stream 1
ostream cout = ostream(&cout_file);

char cerr_buf[1];
filebuf cerr_file = filebuf(2,cerr_buf,0);       // UNIX output stream 2
                                                 // 0-length => unbuffered
ostream cerr = ostream(&cerr_file);
```

This code appears in the source for the stream I/O part of the C++ standard library, not in the ⟨stream.h⟩ header file. The C++ compiler/linker/loader is smart enough to figure out that the ostream constructor needs to be called for cout and cerr before main() is executed. Every static object of a class with a constructor in a program is handled in this way; this is not a "special feature" for I/O.

A destructor is a function that must be executed when an object of its type is destroyed (for example, when an object goes out of scope). The name of the destructor for a class is the complement operator ~ followed by the name of the class, for example

```
~ostream();
```

A destructor often complements a constructor by cleaning up a data structure initialized by the constructor. For example, the destructor for ostream flushes the buffer:

```
ostream.~ostream()
{
        flush();
}
```

In the case of a static object like cout the destructor is called after the execution of main().

The C standard I/O structures stdout and stderr also contain buffers that need flushing. This is done by "magic" in the code that execute a C program since C does not have a general feature for allowing a user to associate initialization and finalization code with data objects. Early C++ implementations could not cope with destructors for static objects and resorted to similar non-general "magic" to implement output streams.

Output of user-defined types

Consider a user-defined type:

```
class complex {
        double  re, im;
public:
        complex(double r = 0, double i = 0) { re=r; im=i; }

        friend  double  real(complex& a) { return a.re; }
        friend  double  imag(complex& a) { return a.im; }

        friend  complex operator+(complex, complex);
        friend  complex operator-(complex, complex);
        friend  complex operator*(complex, complex);
        friend  complex operator/(complex, complex);
        // ...
};
```

Operator << can be defined for the new type complex like this

```
ostream& operator<<(ostream&s , complex z)
{
        return s << "(" << real(z) << "," << imag(z) << ")";
};
```

and used exactly like a built-in type:

```
complex x(1,2);
// ...
cout << "x = " << x << "\n";
```

Note that defining an output operation for a user-defined type does not require modification of the declaration of class ostream, or access to the (hidden) data structure maintained by it. The former is fortunate since the declaration of class ostream resides among the standard header files to which the general user does not have write access. The latter is also important since it provides a good protection against accidental corruption of that data structure. It also makes it possible to change the implementation of an ostream without affecting user programs (see the acknowledgements).

Formatted output

So far << has been used only for unformatted output, and that has indeed been its major use in real programs. There are, however, a few formatting routines that create a string representation of their argument for use as output. Their (optional) second argument specifies the number of character positions to be used.

```
char* oct(long, int =0);        // octal representation
char* dec(long, int =0);        // decimal representation
char* hex(long, int =0);        // hexadecimal representation

char* chr(int, int =0);         // character, chr(0) is the empty string ""
char* str(char*, int =0);       // string
```

Truncation or padding will be done unless a zero-sized field is specified; then (exactly) as many characters as needed is used. For example:

```
cout << "oct(" << oct(x,6) << ") = hex(" << hex(x,4) << ")\n";
```

One can also used a printf style format string:

```
char* form(char* ...);          // printf format
```

Using form() one gets exactly the facilities and problems well known from use of printf(); it is actually sprintf() in disguise. Work is needed to get a satisfactory facility for providing formatted output of user-defined types without the elaboration and type-insecurities associated with the printf approach. In particular, it is probably necessary to find a standard way of providing the

output function for a user defined type with information allowing it to determine space limitations, expectations about padding, left or right adjustment, etc., as expressed by its caller. A practical, but not ideal approach, is to provide functions for user defined types that, like the formatting functions above, produce a suitable string representation of the object for which they are called. For example:

```
class complex {
        float re,im;
public:
        // ...
        char* string(char* format) { return form(format,re,im); }
};
// ...
cout << z.string("(%.3f,%.3f)");
```

Input of built-in types

Input can be done using an `istream` defined analogous to an `ostream`. Here is a small complete program that reads in a number using the operator `>>` ("get from") on the standard input stream `cin`. The number read is assumed to be a number of inches and the program prints the equivalent number of centimeters:

```
#include <stream.h>

main()
{
        int inch;
        cout << "inches=";
        cin  >> inch;
        cout << inch << " in = " << inch*2.54 << " cm\n";
}
```

This can be compiled and run given the input `10` like this:

```
$ CC inch.c
$ a.out
inches=10
10 in = 25.4 cm
$
```

Note that the address-of operator `&` does not appear anywhere in the example above. How then did the number read get into the variable `inch`? The answer is that the input operations are defined using reference arguments. A reference is an alternative name for an object. For example, given an integer

```
int i;
```

we can give it a new name `r` like this:

```
int& r = i;
```

The type `int&` is read "reference to `int`" and use of a reference is synonymous to use of the name of the object it was initialized with. For example:

```
i = 7;
r = 7;
```

both assign 7 to `i`. By declaring an argument to be of type reference the classical "call by reference" is obtained.

Class istream is defined like this:

```
class istream {
        // ...
public:
        ostream* tie(ostream& s);

        istream& operator>>(char*);                    // string
        istream& operator>>(char&);                    // single character
        istream& operator>>(short&);
        istream& operator>>(int&);
        istream& operator>>(long&);
        istream& operator>>(float&);
        istream& operator>>(double&);

        istream& get(char* p, int n, int ='\n'&0377);  // string
        istream& get(char& c);                         // single character

        istream& putback(char c);
        // ...
};
```

The operator>> input functions are defined in this style:

```
istream& istream.operator>>(char& c) {
        // skip whitespace
        int a;
        // somehow read a character into a
        c = a;  // this assignment affects the caller
}
```

Naturally constructors and destructors are provided for the type istream as they were for the type ostream.

One problem remains about why the example worked as intended: How did the output stream cout figure out that it needed to write the prompt "inches=" before the input operation took place? After all, had input and output been independent there would have been no reason to flush cout's buffer until the end of the program, and the program did not contain an explicit flush(). The stream I/O library uses the standard technique of "tying" an output stream to an input stream. This means that cin knows about cout and executes a

```
cout.flush()
```

before attempting to read characters from its device. The operation tie() can be used to tie any output stream to any input stream. For example

```
cin.tie(mystream);
```

would cause cin to flush the output stream mystream instead of cout.

Consider the functions

```
istream& operator>>(char*);     // string
istream& operator>>(char&);     // single character
```

The first reads a whitespace terminated string into a vector of characters; the second reads a single character into a char.

The functions reading floating point constants also accept plain integers.

Whitespace and raw input

The >> operator functions all skip whitespace characters. Whitespace is defined as the standard C whitespace by a call to `isspace()` as defined in <ctype.h>. On an implementation using the ASCII character set this defines whitespace to be the characters blank, tab, newline, vertical tab, formfeed, and return.

Where it is not a good idea to simply treat any sequence of whitespace characters as a token separator the functions

```
istream& get(char& c);                          // single character
istream& get(char* p, int n, int ='\n'&0377);   // string
```

can be used. They treat whitespace characters like other characters. The first reads a single character into its argument. the second reads at most n characters into a character vector starting at p. The optional third argument is used to specify a character that will not be read. Default, the second `get()` function will read at most n characters, but not more than a line: '\n' will not be read.

Stream states

Every stream has a "state" associated with it, and errors and non-standard conditions are handled by setting and testing this state appropriately. The fundamental reason for choosing this approach over the traditional C approach of returning an illegal value in case of trouble was the desire to treat all types (including user-defined types) in the same way, and for many types there is no possible "illegal value" that can be returned. For example, when reading an `int`, and returning an `int` every possible return value represents a legal value, so there is no way of representing end-of-file.

An `istream` can be in one of the following states:

```
enum stream_state { _good, _eof, _fail, _bad };
```

If the state is `_good` or `_eof` the previous input operation succeeded. If the state is `_good` the next input operation might succeed, otherwise it will fail. If one tries to read into a variable v and the operation fails the value of v should be unchanged (it is unchanged if v is of one of the types "known to" class `ostream`). In other words, applying an input operation to a stream that is not in the `_good` state is a null operation. The difference between the states `_fail` and `_bad` is subtle, and only really interesting to implementors of input operations: In the state `_fail` it is assumed that the stream is uncorrupted and that no characters have been "lost". In the state `_bad` all bets are off.

One can examine the state of a stream like this:

```
switch (cin.rdstate()) {
case _good:
        // the last operation on cin succeeded
        break;
case _eof:
        // at end of file
        break;
case _fail:
        // some kind of formatting error
        // probably not too bad
        break;
case _bad:
        // BAD
        break;
}
```

It might be worth noting that if someone invented a new state so that the test above only handled 4 out of 5 cases the compiler would issue a warning.

For any variable z of a type for which the operators >> and << have been defined a copy loop can be written like this:

```
while (cin>>z) cout << z << "\n";
```

For example, if z is a character vector this loop will take standard input and put it one word (that is, a sequence of non-whitespace characters) per line onto standard output.

When a stream (or a stream operation returning a reference to a stream as in the copy example) is used as a condition, the state of the stream is tested and the test "succeeds", that is the value of the condition is non-zero, (only) if the state is _good. To find out why a loop or test failed one can examine the state.

To copy characters (including whitespace characters) the raw input function get() can be used:

```
char ch;
while (cin.get(ch)) cout<<ch;
```

Input of user-defined types

An input operation can be defined for a user-defined type exactly as an output operation was, but for an input operation it is essential that the second argument is of reference type. For example:

```
istream& operator>>(istream& s, complex& a)
/*
        input formats for a complex; "f" indicates a float:
                f
                ( f )
                ( f , f )
*/
{
        double  re = 0, im = 0;
        char    c = 0;

        s>>c;
        if (c == '(') {
                s>>re>>c;
                if (c == ',') s>>im>>c;
                if (c != ')') s.clear(_bad);    // set the state
        }
        else {
                s.putback(c);
                s>>re;
        }

        if (s) a = complex(re,im);
        return s;
}
```

Despite the scarcity of error handling code this will actually handle most kinds of errors well. The local variable c was initialized to avoid having its value accidentally '(' after a failed operation, and the final check of the stream state ensures that the value of the argument a is changed only if everything went well.

More work is needed on the input operations. In particular it would be nice if one could specify input in terms of a pattern (as in languages like Snobol[3] or Icon[4]) and then just test for

success and failure of the complete input operation. Such operations would naturally have to provide some extra buffering so that they could "restore an input stream to its original state" after a failed pattern-match operation.

String manipulation

Traditionally the functions `sprintf()` and `sscanf()` have been used to do I/O-like operations on character strings. Using streams similar operations can be done by binding an `istream` or an `ostream` to a character vector and then using the associated operators exactly as if the stream was bound to a device. For example, if a vector `buf` contains a traditional zero-terminated string of characters the copy loop presented above can be used to print the words from that vector:

```
char buf[SOMESIZE];
// fill buf
istream ist(sizeof(buf),buf);          // make a stream for buf
char b2[MAX];                          // larger than largest word
while (ist>>b2) cout << b2 << "\n";
```

The terminating zero character is interpreted as end-of-file in this case.

Another use of this would be to read a file into a vector of characters replacing every sequence of whitespace characters with a single space:

```
char buf[SOMESIZE];                    // hopefully large enough
ostream ost(sizeof(buf),buf);
char b2[MAX];                          // larger than largest word
while (cin>>b2) ost << b2 << " ";
```

There is no need to check for overflow of `buf`; its associated stream `ost` knows its size and will go into `_fail` state when it is full.

Another look at the output paradigm

Looking at the examples of output above one might conjecture that "an object should not be printed by some general function, but rather print itself given an output stream and maybe also some formatting information as arguments". In other words the output operator for a type `X` should look something like this:

```
class X {
        // ...
        print(ostream& s = cout, format_type& format = default_format);
};
X obj;
// ...
obj.print(cerr);
```

This does have some appeal, but could not be the basic model in C++ since built-in types like `int` are not classes so that it is not possible to write

```
123.print(cerr);        // illegal
```

Furthermore, this style could easily lead to the verbosity that the `<<` operator style of output was invented to avoid:

```
"x = ".print(cerr);     // illegal and verbose
x.print(cerr);
"\n".print(cerr);
```

There are, however, cases where this "inversion" of the output paradigm becomes necessary†.

† The considerations and concepts involved in this example will be familiar to users of Simula67[1], Smalltalk[2], or C++; but may appear rather strange to others. If so, please have a look at any of these languages: the issues are fundamental.

Consider a class shape providing the general concept of a geometric shape:

```
class shape {
        point center;   // every shape has a center
        // ...
public:
        // ...
        virtual draw();
};
```

A shape can be "drawn", that is, printed on a stream, but the function `draw()` is `virtual`. That is, a separate `draw()` function is provided for each particular kind of shape "derived" from class shape. For example:

```
class circle : public shape {
        int radius;     // a circle has both a center and a radius
public:
        // ...
        void draw();
             circle(point cen, int rad);
};
```

That is, a circle has all the attributes of a shape, and can be manipulated as a shape, but it also have some special properties that must be taken into account when it is manipulated. For example:

```
shape* p;
circle c(point(0,0),10);
p = &c;         // the compiler does not know that *p is a circle
// ...
p->draw();      // somewhere else
```

must draw p as a circle. In other words, the fact that the shape pointed to by p is a circle must be deduced at run time from information stored in each shape.

Now consider providing this facility within the *"ostream<<object"* paradigm for output. Like Smalltalk and Simula67, C++ only provides the run time type resolution necessary to determine the actual type of an object at run time using the *"object.operation(argument)"* paradigm, so an << operation must be "inverted". This is how the inversion can be done:

```
ostream& operator<<(ostream& s, shape* p) {
        return p->draw(s);
}

for (shape* p = slist.first(); p; p=slist.next()) cout<<p;
```

Naturally, the reason for the inversion is to maintain a single (terse) paradigm for output operations. Since there is no standard way of passing formatting information on to the `virtual` output function (`draw` in this example), the solution is not perfect.

Buffering

The I/O operations have been specified without any mention of device types, but not all devices can be treated identically with respect to buffering strategies. In particular, an `ostream` bound to a character string needs a different kind of buffer from an `ostream` bound to a file. There is also a need for double buffering of streams connected to network facilities. These problems are handled by providing different buffer types for different streams at the time of initialization (note the three constructors for class `ostream` presented above). There is only one set of

operations on these buffer types, so the ostream functions do not contain code distinguishing them. However, the functions handling buffer underflow and overflow are virtual. This is sufficient to cope with the buffering strategies needed to date, and an excellent example of the use of virtual functions to allow uniform treatment of logically equivalent facilities with different implementations. The declaration of a stream buffer in ⟨stream.h⟩ looks like this:

```
struct streambuf {                  // a buffer for streams

    char* base;                     // beginning of buffer
    char* pptr;                     // next free byte
    char* gptr;                     // next filled byte
    char* eptr;                     // first byte following buffer
    char  alloc;                    // set if buffer is allocated by "new"

    virtual overflow(int c =EOF); // Empty a buffer.
                                    // Return EOF on error
                                    //        0 on success

    virtual int underflow();        // Fill a buffer
                                    // Return EOF on error or end of input
                                    //      next character on success


    int snextc()                    // get the next character
    {
        return (gptr==pptr) ? underflow() : *++gptr&0377;
    }

    // ...
};
```

Note that the pointers needed to maintain the buffer are specified here so that the common "per character" operations can be defined (once only) as maximally efficient inline functions. Only the overflow() and underflow() functions need to be implemented for each particular buffering strategy. For example:

```
struct filebuf : public streambuf {     // a stream buffer for files

    int fd;                         // file descriptor
    char        opened;             // non-zero if file has been opened

    int overflow(int c =EOF);       // Empty a buffer.
                                    // Return EOF on error, 0 on success

    int underflow();                // Fill a buffer.
                                    // Return EOF on error or end of input
                                    //        next character on success
    // ...
};
```

Efficiency

One might expect that since this I/O facility is defined using generally available language features, it is noticeably less efficient than a built-in facility. This does not appear to be the case. Inline expanded functions are used for the basic operations (like "put a character into a buffer"), so the basic overhead tend to be one function call per simple object (integer, string, etc.) written (or read) plus one function call per buffer overflow. This does not appear to be fundamentally different from other I/O facilities dealing with objects at this level.

Conclusion

It is possible to provide an I/O facility using general language level "data abstraction" facilities in such a way that it compares favorably with a built-in I/O facility in both convenience of use, flexibility, and efficiency. The I/O facility presented here provides a single type secure paradigm for input and output of both built-in and user-defined types. Operator overloading (surprisingly) turned out to be an important tool for avoiding the verbosity traditionally associated with extensible type-secure I/O schemes. The I/O facility handles a range of buffering strategies elegantly, and new strategies can be trivially added. It also needs a bit more work, especially in the areas of formatted output and pattern matching input.

Acknowledgements

Many people have helped me in the design of the stream I/O library, and in the design of the C++ language facilities used in their implementation, notably Brian Kernighan, Doug McIlroy, and Jon Shopiro. A special thanks to Dave Presotto who one day got so disgusted with the efficiency of my original implementation that a new and better one was suddenly in place next morning. That is the implementation described above. In accordance with the best theories of data hiding and data abstraction the user interface had remained constant despite a radical change in the implementation strategy.

References

[1] G.Birtwistle et.al.: SIMULA BEGIN
Studentlitteratur, Lund. 1973.

[2] A.Goldberg and D.Robson: SMALLTALK-80 The language and its implementation
Addison Wesley. 1983.

[3] R.E.Griswold et.al.: The Snobol4 Programming Language.
Prentice-Hall. 1970.

[4] R.E.Griswold and M.T.Griswold: The ICON Programming Language.
Prentice-Hall. 1983.

[5] B.W.Kernighan and D.M.Ritchie: The C Programming Language.
Prentice-Hall. 1978.

[6] Bjarne Stroustrup: Data Abstraction in C.
AT&T BLTJ vol 63, no 8 October 1984 pp 1701-1732.

[7] Bjarne Stroustrup: The C++ Programming Language - Reference Manual.
AT&T Bell Laboratories CSTR-108. January 1984.

[8] Bjarne Stroustrup: The C++ Programming Language.
Addison Wesley. To appear fall 1985.

SunNet

JoMei Chang

Sun Microsystems, Inc.
Mountain View, CA 94043
sun ! jmc

Abstract

We propose a layered approach for providing new network services. In this approach, common features among network services are extracted and provided as basic primitives. New network services can then be built upon existing primitives.

The top layer of SunNet consists of high level network services such as a distributed mail service, a distributed time service, and a distributed file service. The middle layer of the SunNet consists of a distributed name server that enables network services to be provided in a location, replication and failure transparent fashion. The lower layer of SunNet consists of two primitives: the status monitor and the distributed synchronizer. These are two powerful primitives for simplifying distributed systems design, and they are the building blocks used to construct the distributed name server.

1. Introduction

This paper describes the architecture of *SunNet: Sun's network services environment.* By network services, we refer to any type of distributed applications that allow workstations to share resources. We envision the future network services in a workstation network environment will include: *a distributed name service, a distributed time service, a distributed file service* (a service that replicates files to increase availability and that provides concurrency control for multiple user access), and *a distributed mail service.* These are examples of services that we believe are important in a workstation network environment. The design goal of SunNet is to identify the common features that are needed by most network services and provide them as basic primitives. This set of basic network primitives can then be used as building blocks to build applications.

The following criteria are used while designing SunNet:

Fault-tolerance. Services should be reliable and fault-tolerant so that a single server failure does not interrupt normal services. The need for reliable services requires the use of replication: each service in SunNet is replicated at a number of sites. Therefore, when a server crashes; the service is still available through other servers. This also provides better performance, as service requests can be spread across all available servers rather than centralized in a single server.

Transparency. SunNet services should be provided in a location, replication, and failure transparent fashion. The transparency requirement calls for the existence of a distributed name server. In SunNet, the distributed name server forms the basis for many other services. For example, a distributed mail server uses the distributed name server to store its mailing list, mail delivery route for each user, and the location of other mail servers.

Scaling. Services in SunNet should be suited for a small network environment (a few workstations) as well as a large network environment (several hundred workstations). A service developed for a small network often fails to function efficiently when it is used in a large network. For example, the *rwho* (remote who) daemon is a function that detects the status of machines in the local net. At each site, once a minute, the *rwho* daemon broadcasts a packet to announce its load and users on the system. On receiving the packets sent by other *rwho* daemons, the daemon stores this information on disk for use in future user queries. While *rwho*

performs satisfactorily in a network with 9 or 10 sites, it fails miserably in a network with over 100 machines. With *rwho* running, the high collision rate, ranging up to 50%, prevents any work to be accomplished within a reasonable time frame.

Expandability. The set of SunNet services should be easily expandable.

To satisfy these criteria, we propose a layered approach (see Figure 1). The top layer of SunNet consists of high level network services. The middle layer of the SunNet consists of a distributed name server that enables network services to be provided in a location, replication and failure transparent fashion. The lower layer of SunNet consists of two primitives: the status monitor and the distributed synchronizer. These are two powerful primitives for simplifying distributed systems design, and they are the building blocks used to construct the distributed name server.

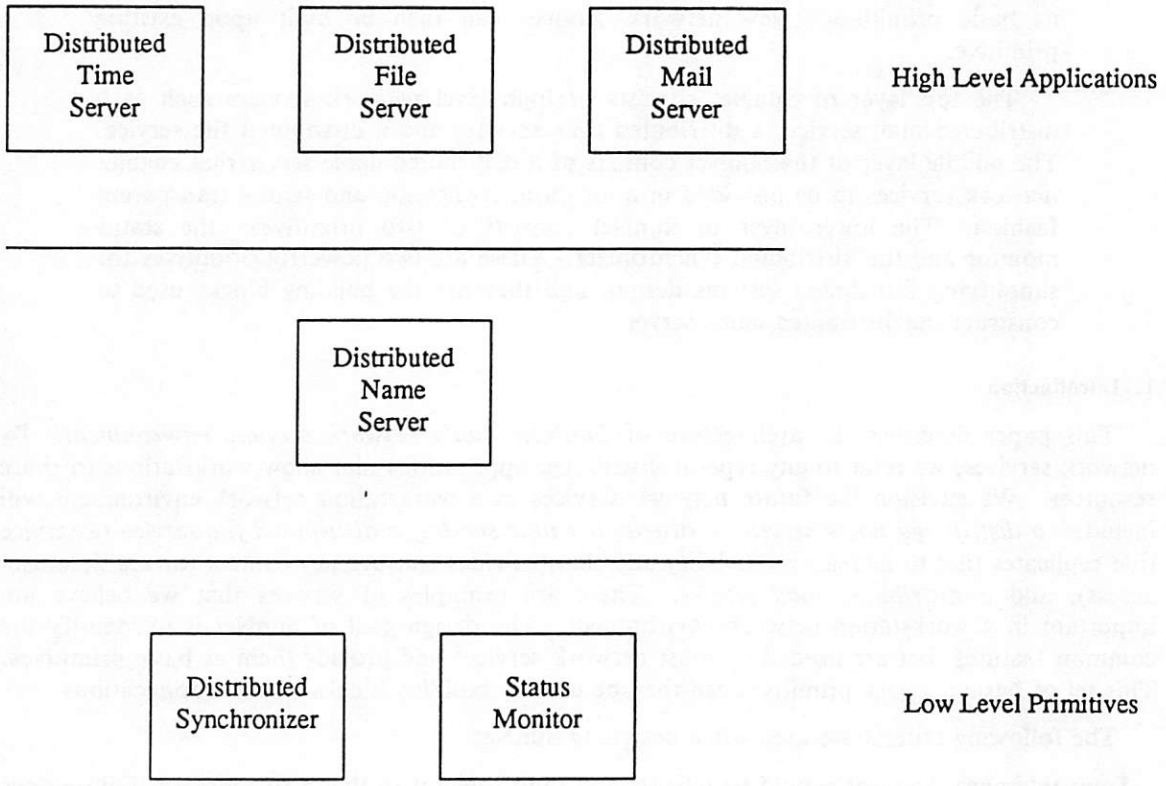


Figure 1. Layered SunNet Approach

2. Environment

A workstation network environment usually consists of a network of workstations —with a high percentage of these machines being diskless — connected by local area networks. The diskless workstations access data located at its file server across the network. In SunNet, the basic access mechanism is provided through the Sun Network File System —Sun NFS [1].

Sun NFS. The Sun NFS permits transparent sharing of file systems in a network of machines. A *server* is a machine that provides resources (exports its file systems) to the network. A *client* is a machine that accesses server resources over the network. The Sun NFS uses a *stateless* protocol. A server does not have to remember from one interaction to the next about its clients and the operations performed. The major advantage of a stateless server is robustness in the face of client or server *crash* (where all processing is halted and the processor state is lost), or in the

face of a network failure. Should a client crashes, it is not necessary for a server to take any action to continue normal operation. Should a server crash or the network fail, it is only necessary that clients continue to attempt to complete NFS operations until the server or network is fixed.

Statefull Services. While Sun NFS provides location transparency and failure transparency features, it does present the following problem. The basic types of file operations such as "read" and "write" can be provided by a stateless server, for certain operations, such as locking, remembering state information is necessary. Statefull services need to be implemented on top of the stateless server.

For example, a lock manager at the server site can be implemented to maintain a local lock table and keep track of local locking information. A lock request from a process at the client machine is first processed by a client lock manager. The client lock manager then forwards the lock request to the server lock manager. Finally, the server lock manager places a lock on the object.

When a service requires resources held by the server, a crash or recovery of either a client or a server interrupts the normal services provided. If the client requests the server to hold a lock (as in the example above), the following undesirable situation can occur. If the client has crashed, the lock can be held forever at the server site. If the server has crashed, the server lock manager loses its state (all the locking information). An essential step in solving this problem is to provide a mechanism to detect site failures and recoveries. (A site is considered as failed if it fails to behave as expected, e.g., fails to send back a response within a predefined time period. A precise failure and recovery definition will be given in the next section.) Then, if the failure of a client is detected, the server releases the failed client's lock. If the server fails, the following procedure, called the *grace period recovery protocol,* is used to reconstruct the server's state information at recovery time.

1. There are two types of requests: normal requests and reclaim requests. When a client lock manager first receives a lock request from a local process, it sends a normal request to the server lock manager. On detecting a server's recovery, the client lock manager automatically sends a reclaim request to the server lock manager. (The reclaim request simply repeats the previous requests).
2. When a server recovers from a failure, the server lock manager waits for a "*grace period*" before it grants any normal requests. During this grace period, any reclaim request will be granted immediately. After the grace period is over, the server lock manager starts processing normal lock requests.
3. When the grace period is over, the server manager reconstructs its state information from the received reclaim requests. Note that the reclaim requests are only from from those client processes that are still interested in its service. If a client process is no longer interested in this service, it can cancel its original request and no reclaim request will be sent to the server lock manager when the server recovers.

This general recovery strategy can be applied to any statefull service. It depends on the existence of a mechanism to detect both client and server failures and recoveries. This mechanism is called a *status monitor.*

3. System Primitives

3.1 Status Monitor

Many distributed applications require the knowledge of the status (up or down) of workstations in the network. In some applications, such as distributed database systems, it is crucial for all sites to agree on each site's status. However, reaching consensus in a distributed environment on site status is a very difficult task.

In SunNet, this task is undertaken by the status monitor. The status monitor guarantees that all sites have a consistent view of site status. Using the status monitor, any status change in the local net, even though a site may crash only for short moment and then recover, can be detected by any other site that is interested. (Note that whenever a site fails, state information is lost and a proper recovery procedure must be invoked.) Providing status monitoring as a basic service, not only frees each application from individually monitoring status of other sites, but also reduces network traffic.

It is important that the overhead of running the status monitor be kept to a minimum. It is also important that an unstable site should not affect the performance of other sites in the network. As the number of sites increases, the complexity of designing a status monitor that satisfies these requirements increases.

Divide and Conquer. The major problem with the *rwho* algorithm is that all the network traffic occurs simultaneously. In a large network, sending a broadcast message to all sites is a very dangerous act. Each time such a message is sent, all receiving sites page in the program from disk (if the program is not in core) to receive the message and, in the case of *rwho*, store the data onto disk. When a high percentage of the machines are diskless, a single broadcast message causes simultaneous network traffic to the file servers and results in very a high collision rate. If responses are expected by the sender, simultaneous responses arriving at the requesting site again cause very high collision rate and possibly lost messages.

In designing the status monitor, we take the divide and conquer approach to handle the problem of monitoring a large network. We choose M sites to be the status monitor server sites, where M is in the range of 4 to 10. A monitor server is usually a more powerful and stable machine with a disk. A file server is always chosen as a monitor server. The remaining sites in the net are status monitor clients. The client sites are divided into M client groups. Each client group is monitored by one monitor server.

(The divide and conquer approach can also be used to monitor site status in an *internet* environment. At each local net, some of the monitor servers are chosen as *internet servers* which handle internet requests. The status of sites at net *i*, are available to the internet environment through the internet servers at net *i*. This is discussed more later.)

The status monitor service is implemented as a set of monitor processes, one at each of the sites that subscribe to the service. A monitor process exists as long as its machine is alive. A server periodically probes the client sites for their status. (This is accomplished by sending a multicast message —a *status message*— to the client group. The content of the status message will be explained later.) The client site only responds when being probed. The monitoring among the servers are handled by forming all server sites into a monitor loop. Each server is

---

* To provide reliable service, the internet service may be replicated at more than one site.

monitored by the next server in the loop. Periodically, a server sends a status message to the next site to indicate it is still alive. When a failed server is detected, the next server in the loop assumes the failed server's monitoring responsibility. In the following, we use *monitored site* and *monitoring site* to refer to the monitoring relationship between the server and a site in its client group and to the monitoring relationship between two consecutive servers in the monitor loop.

Failure Detection. The status monitor declares site *i* *failed* (down) if a monitor process cannot solicit a response from the monitor process at *i* after *R* attempts. The status monitor declares site *i* recovered (up) if a monitor process reestablishes communication with the monitor process at *i*. The status monitor declares site *i* failed and now recovered if between the two times that a monitor process communicates with *i*, site *i* has crashed and then recovered. Note that a slow site, though not crashed, may be declared as failed. A recovery procedure must be invoked at the slow site when communication is reestablished. Although declaring a slow site as failed is costly, it does not affect the correctness of the proposed recovery procedure.

Whenever a monitored site sends back a response, the monitoring site needs to determine whether the monitored site has crashed in between the two probes. This is done by using a state number that counts the number of failures and recoveries that have occurred at the monitored site. Both the monitored site and the monitoring site remember the the monitored site's state number. The monitored site always includes its state number in its response to the monitoring site. However, this number is never stored in the stable storage. Therefore, if a site crashes, it loses its state number. When it recovers and is probed by a monitoring site, an incorrect state number is given in response. A crash is detected by the mismatch of the state numbers.

Failure Announcement. The status monitor uses the site status message to propagate client status change. The status message consists of the state numbers of all sites subscribing to the status monitor service. Each server is responsible for maintaining the state information of its client group. Each time a server detects a client status change, it increments the client site state number. When a server *i* receives a status message from server *i*—1, it adapts any new information from the message except information about its own client group. When the status message is next sent out to the client group or to the next server, the updated state information is propagated out. It takes at most T1 * (M —1) + T2 seconds for the client status change information to be propagated to all sites, where T1 is the time period that a probe message is sent between the servers, T2 is the time period that a probe message is sent between the server and client sites, and M is the number of monitor servers.

Note that status messages are used both for failure detection and for failure announcement. No extra messages are used to announce changes in client status. Therefore, an unstable site will neither cause the status monitor to send extra messages in the network nor affect the performance of other sites.

Failure Synchronization. In the status monitor, failure announcement for a server site is handled differently than the announcement for a client site. There are two reasons for this.

First, the failure or recovery of a server needs to be immediately announced to all sites. In the *grace period recovery protocol,* it is crucial for a client to be immediately notified of the server's recovery so that reclaim requests can be submitted in time. To propagate the server's status change will cause some delay in the client issuing reclaim requests. The delay in announcing the client's status change, on the other hand, is acceptable, since it only affects how soon the server releases the failed client's lock.

Second, if two servers A and B fail simultaneously, and the status change of servers is propagated out through the status message as in the client site case, it is possible that a site i views A as failing before B while another site j views B as failing before A. In distributed database applications, it is crucial to the correctness of the database to ensure that the same sequence of site failure is viewed by all sites participating in a distributed transaction [4]. A consistent sequence of file server failures is therefore important for distributed database types of applications.

In the status monitor, a two phase protocol is used to synchronize server failure announcements. The details of this protocol can be found in [2]. A brief outline of the protocol follows. During phase I of the protocol, any site that detects a server status change, sends out a message to all servers to test their status and to ask them to join its reformation group. An operational server will respond positively to this message if it has not yet joined another group. Otherwise, it responds negatively. The protocol enters phase II; when each server is given R opportunities to respond. A reformation group is successfully formed if all responses are positive. This new group is then announced to all servers and is used as the new monitor loop. Each server then immediately announces the new server status to its client group. Otherwise, the group is aborted and group members are released from the group. A member from an aborted group waits for a random period before it restarts this reformation process.

A monitor failure does introduce some overhead to other servers in the network. However, the servers are chosen from the more stable, better administered machines such as file servers. A monitor server failure should be a rare event.

Internet Monitoring The above procedures describe the status monitoring in a local net. The status monitor can be extended to monitor process status and to monitor site status in an internet environment. The details of these extensions are beyond the scope of this paper. The main principle used in designing the internet monitoring is briefly discussed as follows.

A straightforward approach to internet monitoring is to include in the monitor loop all sites of interest in the remote networks together with all sites of interest in the local network. This solution is not appealing because it causes too much internet traffic. Our approach to internet monitoring is to use *internet servers* to handle all internet requests. All requests to monitor a site located at a remote net i are sent to the internet server at net i. Because the internet server at net i is aware of all changes in site status at its own local net (due to the basic status monitor service), when the status of the monitored site changes, the internet server will notify the site requesting the monitoring service.

The use of internet server reduces the amount of internet traffic. To make this service fault tolerant, internet server failures must be detected and proper recovery procedures must be initiated.

3.2 Distributed Synchronizer

A distributed synchronizer service is provided by using the *reliable broadcast protocol* [3]. The reliable broadcast protocol provides atomic broadcast/multicast. The operation of atomic broadcast ensures two very strong properties:

Atomicity (all-or-nothing property): if a broadcast/multicast message is received by any site in a broadcast/multicast group, it will be received by all operational sites in the broadcast/multicast group;

Message Synchronization: if message Mi is received at one site before Mj, Mi is received at all operational sites in the broadcast/multicast group before Mj.

Atomic broadcast and failure detection have been shown to be two very powerful primitives that simplify distributed system design [4]. For example, atomic broadcast is very useful in maintaining the consistency of replicated data copies in distributed database systems: update requests can be atomic broadcast to all sites holding replicated data copies, and updates are thereby guaranteed to be applied at every site in the same sequence. Atomic broadcast can also be used to simplify the design of a distributed name server, it can be used to provide a network-wide unique number, and it can be used to maintain consistent system internal data among distributed servers. In SunNet, the status monitor provides the failure detection service and the distributed synchronizer provides the atomic broadcast service. The combination of these two basic services forms the lower layer of SunNet.

In the SunNet implementation, atomic broadcast can be implemented very cheaply by exploiting the broadcast features in local area networks. The cost of the atomic broadcast is one acknowledgement per broadcast message (in contrast to one acknowledgement per receiver per message) and is independent of the number of sites in the broadcast/multicast group. To be able to provide such an important service at such a low cost makes SunNet an attractive basis on which to build distributed applications.

3.3 A Distributed Name Server

The name server maintains a number of name to content mappings. To provide reliable service, the name server files are replicated at multiple sites. Three types of consistency are provided by the distributed name server to ensure mutual consistency among the name server files.

Level 1. Level 1 consistency guarantees that all name server files are consistent at all times. It is provided by using the distributed synchronizer as described above.

Level 2. Level 2 consistency allows two copies of name server files to be temporarily inconsistent while an update made to one copy is being propagated to another copy. This is the type of consistency provided in the Xerox Grapevine [5] system.

Level 3. Level 3 consistency only synchronizes the replicated copies of the name server files periodically, e.g., every 24 hours.

We believe for most applications level 2 and 3 consistency are sufficient. Initially, level 1 consistency will be reserved for system internal usage. The distributed name server again forms the basis of other distributed services. It provides the replication, location, transparency for those services and provides a simple mechanism for data storage and retrieval.

4. Status

The current status of the SunNet project is summarized as follows. The status monitor, described in Section 3, has been implemented to monitor site status in a single local net — Sun's engineering net — which consists of over 50 sites. The implementation of the status monitor in the internet environment is currently in progress. The implementation of the distributed synchronizer and the implementation of the distributed name server will begin shortly.

5. References

1. R. Sandberg, "The Design and Implementation of the Sun Network File System" Proc Usenix, June 1985.
2. J. M. Chang, "Status Monitor", paper in preparation, 1985.
3. J. M. Chang and N. F. Maxemchuk, "Reliable Broadcast Protocols", ACM Transactions on Computer Systems, August 1984.

4. J. M. Chang, "Simplifying Distributed Database Systems Design by Using a Broadcast Network", Proc ACM SIGMOD, June 1984.

5. A. D. Birrell, et. al. "Grapevine: An Exercise in Distributed Computing", CACM, April 1982.

Project Stargate

Lauren Weinstein

Computer/Telecommunications Consultant
P.O. Box 2284
Culver City, California 90231
(213) 645-7200
UUCP: {ihnp4, decvax, clyde, bonnie, seismo}!vortex!lauren

This is an update regarding the ongoing progress of "Project Stargate," an experimental system for the transmission of "Usenet netnews"-type messages via satellite. The experiment, which is currently sending test messages over the vertical broadcast interval of satellite television "Superstation" WTBS (based in Atlanta) has now been actively operating for nearly six months, and is proceeding quite nicely.

For details regarding the general topic of television vertical interval data transmission systems, please see my paper in the June, 1984 (Salt Lake City) Usenix Conference Proceedings [1]. Details regarding the genesis, installation, and operation of Stargate itself are presented in my paper from the January, 1985 (Dallas) Usenix Conference Proceedings [2].

The Project Stargate experiment has been able to proceed thanks to various firms which have made money, hardware, and/or services available for our use. Primary among these include Usenix, Southern Satellite Systems (the satellite carrier for WTBS), Fortune Systems, and Bell Communications Research. More details regarding the involvement of these organizations, and of the many individuals who have been actively supporting the project, can be found in the Dallas conference paper referenced above. Without the help of the persons at these firms and other individuals who are supporting this volunteer work, my original conception for Stargate could never have proceeded beyond the idea stage into the realm of reality. I'd like to take this opportunity to thank them all for their support!

Progress has been continuing on the project in both the organizational and technical areas. Usenix, as the primary sponsor of the project, has now formed a "Stargate Subcommittee" to study the various organizational and legal aspects of the project. The members of the subcommittee include Lou Katz (EECS Department, U.C. Berkeley), Mike Lesk (Bell Communications Research), Brian Redman (Bell Communications Research), Stu Feldman (Bell Communications Research), Steve Johnson (AT&T Bell Laboratories), and myself. The subcommittee is jointly co-chaired by Lou Katz and Mike Lesk.

The subcommittee is working to establish a functional framework for both the continuation of the experiment itself and for its potential evolution into a practical, reliable data transmission service. The entire range of related issues is under consideration, including specific organizational frameworks, financing, operations, information content, and many other topics. Since the subcommittee has only recently begun its organized efforts in this area, it's too early to report any specific results--but the work is underway. I hope to be able to report in more detail regarding the subcommittee's efforts in the near future.

The experiment itself continues well. A collection of test messages continues to cycle continuously through the Stargate data stream to the test data decoder (receiving WTBS via a local cable television outlet) at my location. At the Dallas Usenix conference, a successful demonstration of Stargate was held via a small, rented satellite receiver which received WTBS directly from the appropriate satellite. I've been continuing my testing of data reception quality at various locations (most recently at Lucasfilm, Ltd. in San Rafael, California). All test results to date have been

extremely encouraging.

The experiment is now proceeding to a broader phase. Up to now, only two of the required test data decoders have been available for my use with Stargate. By the time of the actual presentation of this paper at Portland, however, I hope to have approximately six additional decoders available for placement (or already placed) at a variety of locations for an interval of continued testing. A substantial number of organizations have already expressed interest in receiving these decoders (on a free loan) and I expect to have them all allocated well before the Portland Usenix conference itself.

Additionally, I plan to begin (before the start of the conference) experimental feeds of current messages from the "mod.*" (moderated) Usenet netnews groups via Stargate to replace the repeating test messages that have been circulating up to now. It should be noted that all of these efforts are still only an experiment, not a functional "service" by any means. Any data being transmitted represents convenient input material for the experiment and will not necessarily reflect the sorts of transmitted materials in any actual service--that's impossible to predict at this point.

There's considerable other progress also taking place. The data decoders currently available for our testing are fairly expensive, comparatively bulky units that will not be the same as the smaller, less expensive units that would be used in an actual service. The newer decoders, which are not yet available on a production basis, will include integral error correction, encryption, remote addressing capabilities, and other useful functions. I've been told that the work on obtaining these new decoders (being manufactured by a very large consumer electronics firm) is proceeding well and should not ultimately delay the progress of Stargate in any manner. Also, work has been proceeding on finding design and manufacturing facilities for the Stargate "buffer board/box" which will be needed to take data from the decoder and provide various required functionalities for message processing, organizing, and controlled feeding of data to host computers. While it would be premature for me to discuss the details of the efforts relating to this particular equipment, I will simply say that some very promising possibilities are being explored and I have no reason to expect any significant problems in this regard.

We've planned another demonstration of Stargate for the conference at which this paper will be presented, and I hope to see many of you there. If you have any questions or comments regarding Stargate, please feel free to contact me at the above UUCP network address, or at the above phone number if necessary. I discourage the use of physical mail unless absolutely necessary, since I can respond much more quickly to electronic mail or phone calls.

We're now about one year from my original Usenix conference presentation regarding vertical interval satellite transmission systems. We've come a long way in a short period of time, and everything is still looking excellent for the future of the project. I have high hopes that it will continue to evolve toward becoming a service that will be a practical and economical enhancement of our existing network message transmission and reception capabilities. With the continued support of the Usenix community, we have a good chance of seeing a successful outcome. Once again, thanks very much to all of you who have expressed support for this project and have enabled it to become a reality!

References

[1] Lauren Weinstein, Broadcasting of Netnews and Network Mail via Satellite. USENIX Conference Proceedings, Summer 1984.

[2] Lauren Weinstein, Netnews via Satellite: A Progress Report (12/84). USENIX Conference Proceedings, Winter 1985.

Face the Nation

Rob Pike
research!rob

David L. Presotto
research!presotto

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Digitized images of the faces of members of AT&T Bell Laboratories' Computing Science Research Center, and of many of their electronic mail correspondents, are stored in a network 'face server' accessible from the Eighth Edition machines at Murray Hill. The faces decorate line printer banners, announce the arrival of mail, and perform a variety of other Information Age services. The face server is implemented as a variant of the network file system, so the images are stored on one machine but appear to be regular files in the file systems of all the machines on the network. This method of presenting the information is attractive for databases with inherent structure, as it permits standard Unix system utilities to access, manipulate and maintain the database.

Spurred on by initial successes with some experiments with famous people (E.W. Dijkstra and P.J. Weinberger), we last year undertook to digitize the faces of all people in our research center and of those with whom we communicate. The result of this effort was about thirty megabytes of disk space, which soon began to spread to other computers that needed access to the faces for announcement of mail, banners on printed output, and so on. This paper presents the design, implementation and use of a network 'face server' that eliminates the need for duplication. The faces are stored in one place that allows all machines and all users to access this treasury of useful information.

Polaroid 4" by 5" black and white photographs of each participant, lit fairly flat with a white background, were digitized by a video frame grabber into 512×512 byte (8 bits per pixel) grey-scale images. Each image was then reduced interactively to a 48×48 bit (1 bit per pixel) black-and-white 'ikon.' The reduction program uses the Floyd-Steinberg algorithm [Newm79]; the interactive activity is setting the upper and lower thresholds to adjust brightness and contrast, then selecting the best-looking image. Although the small pictures are surprisingly recognizable, the quality of an ikon depends critically on the parameters. It seems to need human judgement to choose the best digitization. After the first day's photography session we had over a hundred faces recorded, and it became obvious we needed a central place to store, maintain and administer the face files — a network face server.

The faces were originally stored as ordinary files in a single directory, so the tools to access them were standard programs such as `cat(1)`. But instead of a simple list of files, the structure we wanted was hierarchical, with users of a particular machine, or machines in an organization, grouped together. The obvious way to construct this hierarchy was in the file system, with links to

associate faces (files) with the many machines (directories) they inhabit. To make the file system available everywhere, the face server simulates a Unix† file system that can be mounted on the client machine using the Eighth Edition network file system protocols. Faces are named by ordinary file names, so no special software is required to use them. (Pike and Weinberger [Pike85] discuss other issues related to this naming scheme.) The details of the file system are different, however, to match the structure of the database and its typical uses.

By contrast, the traditional model of a network server is fairly complicated and different from our usual methods of accessing data. A typical server involves operations such as 'attach protocols' and 'transactions' instead of system calls such as `open`(2) and `read`(2). Inventing new methods of accessing data requires inventing new software to use the methods, but the standard system calls provide all the functionality needed.

Our face server has two file formats. The small ikons are stored as ASCII hexadecimal strings as they would appear in a C declaration, while high-resolution grey-scale images are stored as binary files for storage efficiency. The file system appears as a directory, conventionally `/n/face`, with constituent files named, for example, `/n/face/research/pjw/48x48x1`. The first subdirectory contains machine names, the second users on those machines, and the third the actual files, named by their resolution.

Our first implementation used the regular Eighth Edition network file system [Wein84], which allows a remote system's root directory to be logically mounted on a node in the local file system. A simple early implementation of the face server mounted the root of the server system on the file system of each other machine (the client). This used only existing software but caused problems. First, access protection was not implemented in a manner appropriate to a network service. The network file system made visible all the server machine's files, not just the faces. Also, because of the way the network file system implements protections, anyone wanting to use the faces needed an account on the server machine. These details left us with an uncomfortable choice: either to provide to users on client systems more access than they needed, or to deny certain users or systems access to the face server. Second, resources on the server machine were strained by the implementation. The remote file system requires the server system to maintain an open network connection and a server process for each of its client systems. Since we have dozens of systems, this caused the server system to run out of communications channels, process slots, or swap space. Finally, the desired multiply-connected view of the data was clumsy and inefficient when built with traditional file system techniques: the large number of symbolic links were expensive in both disk space and CPU time to access.

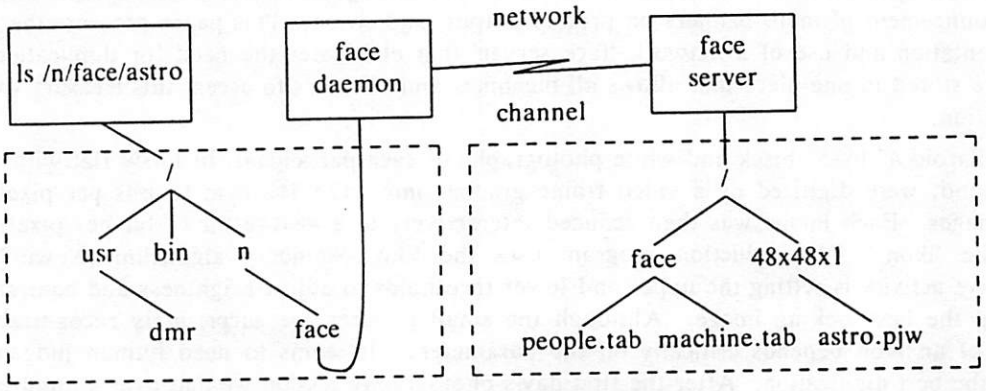


Figure 1. Structure of the face server.

The current implementation avoids these problems by dividing the work of the server between two different processes while leaving the flow of data through the kernel exactly the same as for a regular network file system (see Figure 1). A daemon process runs on each client system

† Unix is a trademark of AT&T Bell Laboratories, for what it's worth.

and a single server process runs on the server system. Each client daemon maintains the state information concerning open face files for its system. These daemons translate file system primitives to remote procedure calls to the server process across the network. Because of the low duty cycle on a connection and the relative scarcity of channels on the service machine, the link drops automatically if it is not used for a few minutes. The server is stateless in the sense that all incoming requests have full information; the server does not remember information about calls. This simplifies the job of creating and shutting down connections, and results in greater reliability and resilience to failures (such as system crashes) than for the regular network file system. The file access problems are solved by making only face files visible through the face server. This makes it possible to apply special protection methods to face files without affecting access to other files on the server system.

The server reads a pair of files (one for machine names, one for people names) that describe the correspondence between machine/person pairs and regular files on disk, and builds an in-core tree structure corresponding to the face file system's directory tree. The leaves of the tree are pointers to the actual files on disk, but the directories are kept in core so directories can be 'hard linked' together (this cannot be done on disk because the link bits in a directory i-node are usurped for directory tree consistency checking); machines related by organization (e.g. mit-eddie and MIT-MC) are linked to a directory that names the organization (MIT). Maintenance consists of updating the description files to reflect the correspondence between the real world and the availability of images.

The face server provides a simple database of faces; there is also a conventional protocol to convert a machine/person pair to a face. The directory for an organization may contain an 'unknown' face to be used when the appropriate user's face is not available, and there is an artificial organization 'misc.' to store ikons for generic users such as root and uucp. The following faces are those retrieved for `research!pjw` (standard available face), `lucasfilm!george` (the unknown face for lucasfilm, `lucasfilm/unknown`) and `decvax!uucp` (the general uucp ikon, `misc./uucp`):

On rare occasions, such as when its host machine has just had preventive maintenance, the face server may be down. For such times, some facial programs keep internally a blank face not stored in the face server, so some face is always available:

The main client of the face server is a program called `vismon` that continually monitors the CPU load on systems on the network and reports the arrival of mail. The sender of each message is converted to a face and displayed in `vismon`'s window (see Figure 2). The result of an afternoon's mail is a police lineup of faces, perhaps with a hand-drawn ruminant.

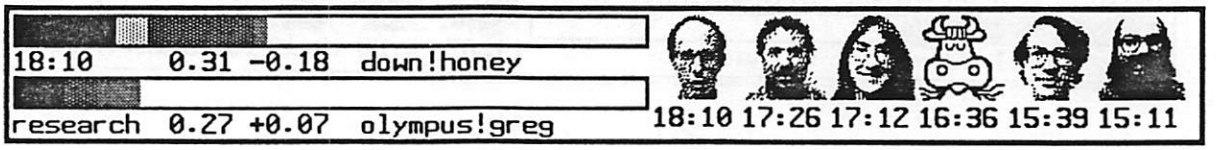


Figure 2. A `vismon` window, showing a collection of faces and the system activity on a couple of machines.

Although small, the images are remarkably good likenesses of their subjects, and because the picture of a given person is the same from day to day, it comes to represent the person strongly; the word ikon is particularly appropriate. (*Chambers 20th Century dictionary* defines ikon as, "a symbol, representation: anybody or anything uncritically admired.") At time of writing we have 263 faces in our database, but only about 60 in our own research center. The other two hundred

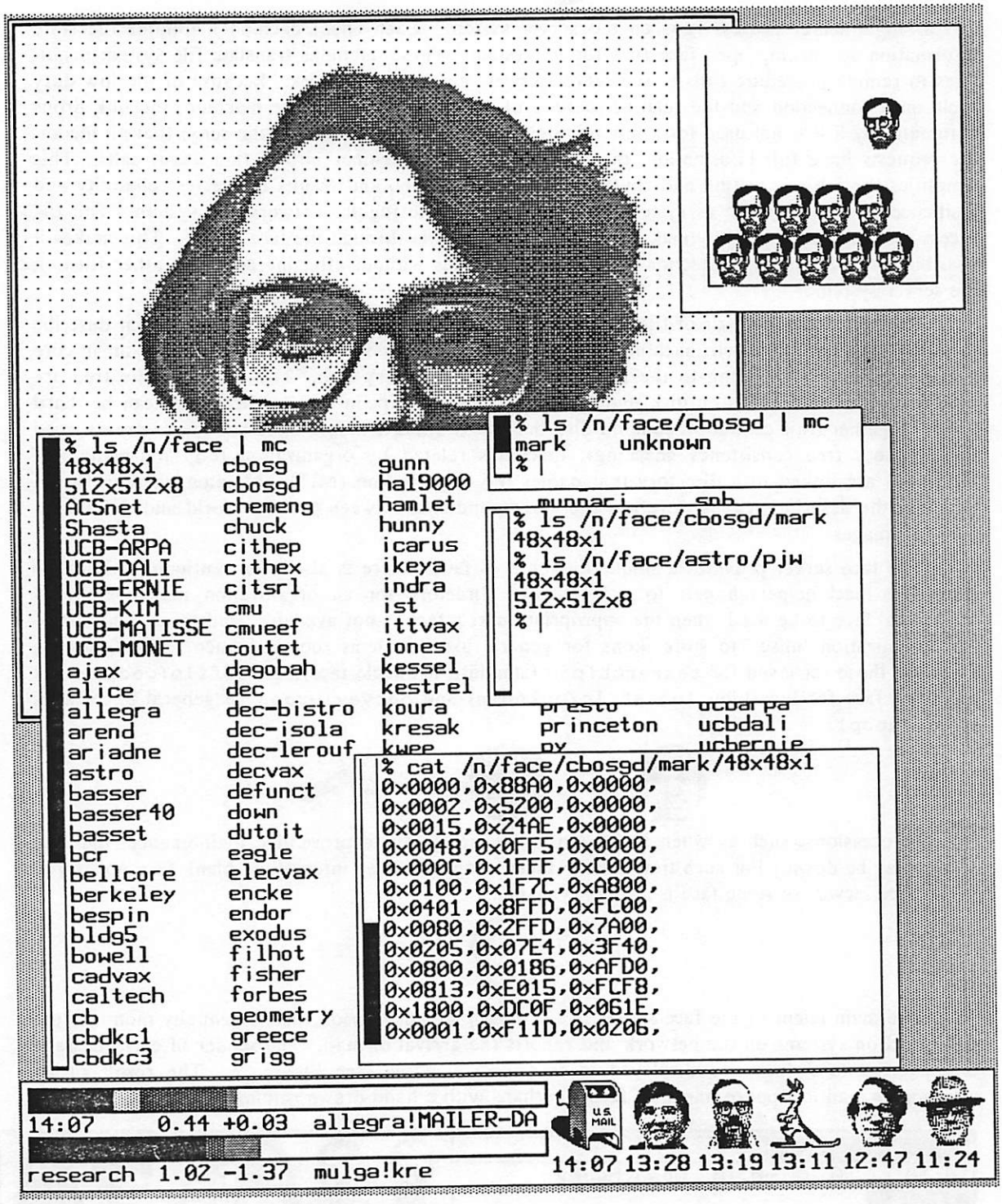


Figure 3. A Teletype DMD-5620 screen displaying the result of an afternoon's activity to create a Figure 3 showing the face server being used. The top two programs show a dithered 512×512×8 face and an early experiment with hand-drawn faces. The central group of programs illustrate the structure of the face file system. The bottom program is a `vismon`, described in the text.

are people outside our group, some even from other continents, and therefore less familiar. Nonetheless, the face images make their owners more a part of the local community; a face seen only once a month is more recognizable than three-letter initials seen daily.

The lessons from the face server involve the method of presentation of the database: as a set of regular Unix files. By providing conventional names for the faces, ordinary Unix tools can be used as database utilities. Less obvious, the behavior of files is so well known that there was very little work to do in building the service; the phrase "file system" determines most of the details. We resisted the temptation to extend the semantics of the files by, for example, creating files of particular resolution on demand or coding the name of the file to determine ASCII or binary format; instead, the face server provides regular Unix files, building on their behavior rather than changing it. The implementation was made simpler by the existence of a working network file system and the Eighth Edition IPC mechanisms. Now that we have one database working, others may appear. One possibility is a digital font server implemented as a file system.

Acknowledgements

Luca Cardelli drew the first few ikons by hand, ran the digitizing hardware and software for the first hundred or so faces, and deserves credit for putting faces on our minds. P.J. Weinberger provided the network file system and an inspiring visage.

References

[Newm79] W.M. Newman and R.F. Sproull, *Principles of Interactive Computer Graphics*, p. 226, McGraw-Hill, New York 1979.

[Pike85] Rob Pike and P.J. Weinberger, "The Hideous Name," *Summer 1985 USENIX Conference Proceedings*, Portland, Oregon.

[Wein84] P.J. Weinberger, "The Version 8 Network File System," *Summer 1984 USENIX Conference Proceedings*, Salt Lake City, Utah.

NAME

faced — network face server

SYNOPSIS

/usr/net/face.go

DESCRIPTION

The network face server provides a database of 48×48 bit icons and other facial representations. It is implemented as a network file system similar to *netfs*(8).

The file system, conventionally mounted on */n/face*, has a fixed three-level hierarchy. The first level is a machine name, the second level a user name, and the third level a resolution. Thus the file */n/face/kwee/pjw/48x48x1* is the standard face icon (for user pjw) on machine kwee:

Many local users also have 512×512 byte high-resolution faces, named *512x512x8*. Other resolutions may also be present for a particular face. One-bit images are stored in the format used by *icon*(9.1); eight-bit images are arrays of bytes. The directories for machines sharing a user community, such as those on a Datakit node, are linked together and given a name appropriate to the community. For example, */n/face/kwee* is a link to */n/face/astro*.

To access the face for a mail name *machine!uid* take the result of the first successful open from the following list of files:

```
/n/face/machine/uid/48x48x1
/n/face/misc./uid/48x48x1
/n/face/machine/unknown/48x48x1
/n/face/misc./unknown/48x48x1
```

The directory *misc.* holds faces for generic users such as *root* and *uucp*. The face server is made available on a machine by running */usr/net/face.go* from *rc*(8).

The face server data is kept on kwee, and is administered by a pair of ASCII files that associate related machines and faces. The machine table *machine.tab* attaches machines to communities; the line

```
kwee=astro
```

puts the machine *kwee* in community *astro*. The people table associates a machine/user pair in the face server with a file on kwee;

```
astro/pjw=pjweinberger
```

causes the images stored in disk files named *pjweinberger* to be available in the face server in directory */n/face/astro/pjw*. As well, each disk file used by the face server is linked (by its original name) into the directory */n/face/48x48x1* or */n/face/512x512x8* for easy access to all the images.

FILES

/n/kwee/usr/jerq/icon/face48	directory of low resolution faces
/n/kwee/t0/face/512x512x8	directory of high resolution faces
/n/kwee/usr/net/face/people.tab	people/file equivalences
/n/kwee/usr/net/face/machine.tab	machine/community equivalences

SEE ALSO

netfs(8), face(9.1), icon(9.1), sysmon(9.1)

BUGS

After updating the tables, an indeterminate time may pass before the new faces are available.
All face server files are unwritable.

The Snocone Programming Language

Andrew Koenig

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill NJ 07974

ABSTRACT

SNOBOL4 is a well-known programming language with convenient semantics and clumsy syntax. Following the pattern of Ratfor and EFL, we have added "syntactic sugar" to SNOBOL4, with an eye toward making it easier to use. We call the result *Snocone.*

This paper describes the Snocone language in enough detail that people not familiar with SNOBOL4 can learn to use it, although previous familiarity with SNOBOL4 will help.

Introduction

The semantics of the SNOBOL4 programming language1 include such unusual and useful features as dynamic typing, a general-purpose garbage collector, excellent character string facilities, associative arrays, and strong run-time diagnostics. Griswold2 Gimpel3 and others have documented many ways of doing things in SNOBOL4 that can only be accomplished with much more difficulty in other languages.

Unfortunately, the control structures of SNOBOL4 are archaic, and the fact that blank is an operator tends to encourage certain types of errors that are hard to detect. Other aspects of the language make it a nuisance to construct large programs out of small parts.

To ameliorate some of these problems, we have designed and implemented a new language that provides some syntactic sugar for SNOBOL4. The obvious name for such a language is *Snocone.*

The design of the Snocone language was inspired by Ratfor4 and EFL5 Like EFL, and unlike Ratfor, Snocone is a self-contained programming language, rather than a proper superset of SNOBOL4. Like Ratfor, and unlike EFL, the Snocone translator makes no attempt to produce SNOBOL4 output that is easy for humans to understand.

Hanson6 has written a similar, but simpler preprocessor for SNOBOL4. His preprocessor is like Ratfor in that it does not change the statements in the basic language but rather adds new syntax.

Griswold7 has written a preprocessor for a language similar to Snocone, the syntax of which is based on Icon8 rather than on C or EFL. His preprocessor is written in C, and uses Yacc for its parsing.

In contrast, Snocone has a syntax roughly based on C and is written in Snocone.

What's nice about SNOBOL4

Variables in SNOBOL4 are dynamically typed: the type of any variable is the type of the value most recently assigned to it. Thus, declarations are unnecessary, and, in fact, SNOBOL4 has no declarations as such (except that procedures can have local variables).

All operators and built-in functions check their argument types; each argument is converted automatically to an appropriate type. A run-time diagnostic message results if a conversion is impossible. For instance, if an addend is a string, SNOBOL4 will try to convert it to an integer or real number, depending on the form of its value. If the value is inappropriate to convert to a numeric type, the program will halt.

Like Lisp, and unlike most other languages, SNOBOL4 does not have any explicit mechanism for returning memory to the system. Rather, the implementation must detect when memory is no longer needed and make that memory available for re-use. This makes the language harder to implement, but easier to use. One nice by-product of this sort of memory allocator is that SNOBOL4 programs tend to be free of arbitrary size limitations.

One SNOBOL4 datatype is the *pattern*, which describes an (arbitrarily complex) set of character strings. Briefly, the language incorporates a general top-down backtracking parser. This parser is so easy to use that it is the usual way of dealing with strings in SNOBOL4.

Another useful datatype is the *table*. A table is like a one-dimensional array, except that its subscripts are not limited to being integers. For instance, a compiler symbol table might be maintained as a SNOBOL4 table, with the subscript being the identifier and the value being some structure which holds the desired information about that identifier.

An interesting and unusual semantic idea in SNOBOL4 is that of *statement failure*. Many operations can easily encounter conditions that preclude their execution. For example, an attempt may be made to access a non-existent array element, read beyond the last record of a file, search a string for a pattern that it does not contain, or even make a comparison that gives an unexpected result. When this happens, the operation *fails*. Usually, the failure of an operation implies the failure of the statement of which it is a part. While statement failure is not an error condition, it can be tested. In fact, conditional transfer on the success or failure of a statement is virtually the only kind of control structure in SNOBOL4.

A good implementation of SNOBOL4 is Macrospitbol, by Dewar and McCann9 This is a portable threaded-code interpreter, which is fast and robust enough to be used for "production" purposes. For example, on a VAX-11/780 it translates about 150 statements per second and executes about 5,000.

SNOBOL4 implementations in general have excellent run-time diagnostic facilities. The language definition includes ways of tracing programs for debugging, and of trapping almost any kind of run-time error.

What's not nice about SNOBOL4

Here is a simple SNOBOL4 program to add the integers between 1 and 1000:

```
        sum = 0
        term = 1
loop    sum = sum + term
        term = term + 1
        LE(term,1000)                :s(loop)
        OUTPUT = "The sum is " sum
END
```

The first two lines of this program assign values to variables. The name *loop* in the third line is a label; it is recognized as such because it begins the line without any white space ahead of it.

The fifth line calls a built-in function to compare the value of the variable *term* to 1000. SNOBOL4 has no comparison operators: all comparisons are done by *predicate* functions that either return the null string if the condition being tested is true or fail if the condition is false. The `:s(loop)` at the end of the line is a conditional **go to**: it causes control to transfer to the label *loop* if the statement succeeds, and to the following statement if it fails.

In contrast, the program looks like this in Snocone:

```
sum = 0
term = 1
do {
        sum = sum + term
        term = term + 1
} while (term <= 1000)
OUTPUT = "The sum is " && sum
```

In the past fifteen years, the trend in programming language design has been overwhelmingly toward programming languages with control structures that make it reasonable to write programs with at most a very few **go to** statements. SNOBOL4 exhibits almost the exact opposite of this trend: essentially the only control structure in SNOBOL4 is a form of conditional **go to** statement. There is no block structure, and all labels are global. Thus, writing a SNOBOL4 program requires a constant effort to invent new label names, and to choose names that have at least some chance of telling the reader whether their use is local or global.

The programmer's job is not made any easier by the peculiar way SNOBOL4 handles subroutines. SNOBOL4 subroutines are defined at run time by a built-in function called `DEFINE`. Its argument is a *prototype* of the subroutine to be defined — a character string that describes how the subroutine is to be used. The subroutine's entry point is the label with the same name. Because the statement bearing this label is not special in any other way, it must be placed where it will not be executed in the ordinary course of events. The call to `DEFINE`, on the other hand, must be executed before the subroutine it defines can be used.

This leads SNOBOL4 programmers into contortions. To keep the `DEFINE` call near the subroutine body, one must invent yet another label:

```
        DEFINE("square(x)")             :(square.end)
square  square = x * x                  :(RETURN)
square.end
```

Alternatively, one can put all the `DEFINE` calls in one place, at the cost of moving the body of each subroutine arbitrarily far from where its prototype appears.

The Snocone programmer has an easier job:

```
procedure square (x) {
        return x * x
}
```

Motivation

Snocone's purpose, then, is to make it easy for a programmer used to a block-structured language like C to write programs that have the freedom and semantic flexibility of SNOBOL4. To this end, we have changed the SNOBOL4 language in several ways.

First, we introduced an explicit concatenation operator. SNOBOL4 uses blank for concatenation, so

```
y = f(x)
```

is a function call, but

```
y = f (x)
```

assigns to `y` the concatenation of the values of the variables `f` and `x`. We chose `&&` to represent concatenation because SNOBOL4 programs often use concatenation for the same purpose that C programs use `&&`.

Second, we allow much the same freedom with spaces that C does, with one exception: newline ends a statement. Statements may be continued onto multiple lines in much the same way as in EFL programs: a line is continued if it ends with an operator or some kind of open bracket.

Third, we have added procedure and structure declarations. These things are accomplished in SNOBOL4 by calling built-in subroutines, and the context of the calls is often obscure.

Finally, we have introduced control structures similar to those in C and other block-structured languages: the `if`, `while`, `for`, and `do` statements.

LANGUAGE DESCRIPTION

Lexical Conventions

Blanks (spaces and tabs, but not newlines) may not appear within a token, but may freely separate tokens. Comments begin with a # character and end at the end of the line.

Constants may be integers, reals, or strings. There are no signed numeric constants. Integers range from 0 to $2^{31}-1$. Real constants are short-precision only, and must contain either a decimal point or an `E` (or `e`). String constants may be delimited by single or double quotes with the same meaning. Characters in quotes receive no special interpretation. A single quote may appear in strings delimited by double quotes, and vice versa.

Identifiers may be of any length. All characters in an identifier are significant, but their case is presently not significant. Case may become significant in the future. An identifier is a non-empty sequence of letters and digits, beginning with a letter. Underscore (_) is a letter. The same identifier may be used independently to represent a procedure, label, or variable.

Statement Separation

Snocone delimits statements much like the Shell10 (the command interpreter in the UNIX† operating system)11 or EFL. A statement is ended either by a semicolon or newline, except that if the last token on a line is an operator or open parenthesis or bracket, the next line is automatically considered as part of the current statement. Newlines may also separate clauses of compound statements, so

```
if (a < 0)
        a = 0
```

is acceptable and means the same as

```
if (a < 0) a = 0
```

or, spreading things as much as possible:

† UNIX is a Trademark of Bell Laboratories.

```
if (
a <
0)
a =
0
```

File Inclusion

The contents of an arbitrary file can be incorporated into the program by writing an *include line* in one of the following four forms:

```
#include "file"
#include 'file'
#include <file>
#include {file}
```

Spaces may appear between # and `include`. The contents of the named file are substituted for the `include` line.

The exact behavior of an `include` line depends on the delimiters surrounding the file name. If quotes are used (single or double), the file is sought in the current directory. If brackets are used (angle brackets or curly braces), the file is sought in an installation-dependent system library (*/usr/lib/snocone* on our systems). If double quotes or angle brackets are used, the file is included unconditionally, even if the same file is included several times. If single quotes or curly braces are used, the file will only be included once, even if it is named in several `include` lines.

This latter behavior is useful in the following sort of situation. Suppose that procedure *proc1* is defined in file *proc1.h*, and procedure *proc2* is defined in file *proc2.h*. A program that calls both *proc1* and *proc2* might then contain:

```
#include "proc1.h"
#include "proc2.h"
```

Now, suppose that *proc1* is changed to call *proc2*. If *proc1.h* is made to contain

```
#include "proc2.h"
```

then our hypothetical sample program will wind up with two copies of *proc2.h* and will therefore run into trouble with duplicate procedure definitions. If, however, *proc1.h* contains:

```
#include 'proc2.h'
```

and the main program contains:

```
#include 'proc1.h'
#include 'proc2.h'
```

then *proc2* will be defined only once.

Expression Evaluation, Success, and Failure

Evaluating an expression has one of three outcomes: a value, failure, or error.

Errors normally result in a diagnostic message and termination of the program, and arise from the usual sorts of things: overflow, underflow, division by zero, impossible conversions, running out of memory, and so on.

Failure, on the other hand, is not an error condition. An expression that fails is merely one that does not yield a value. Examples of expressions that fail include attempts to refer to non-existent array elements, attempts to read beyond end of file, and even comparisons that yield unexpected results. For instance, the expression

```
a < b
```

yields a null string if `a` is indeed less than `b`, and fails otherwise.

Most operators fail if any of their operands fails. The few exceptions will be noted below.

An expression that always either yields a null string or fails is sometimes called a *predicate*. Testing such expressions for success or failure in `if`, `while`, and `do` statements is the primary way of affecting the flow of control in Snocone.

Data Types, Declarations, and Scope

Variables in Snocone are dynamically typed: a variable has the type of the value most recently assigned to it. Except for a few predefined variables (described under *pattern matching*), all variables have the null string as their initial values. All variables are global, but procedures can nominate variables whose values will automatically be saved at entry and restored at exit. The only declarations define structures:

```
struct cons {car, cdr}
```

In effect, this declaration defines three procedures named `cons`, `car`, and `cdr`. The value of `cons(a,b)` is a newly-created `cons` object with `a` and `b` as the values of its `car` and `cdr` fields, respectively. The fields, in turn, are accessed by similarly-named functions. For instance, the `cdr` field of `cons` structure `x` is accessed as `cdr(x)`. Thus this program:

```
struct cons {car, cdr}
a = cons (3, cons (4, 5))
OUTPUT = car (cdr (a))
```

prints 4. The procedures corresponding to field names may be used as the target of an assignment:

```
car (a) = "Hello"
```

Snocone offers other data types than those described above. While numeric constants cannot be negative, variables and expressions suffer from no such restriction. Strings may be of any length up to an implementation-defined upper bound, usually many thousands of characters. All variables have the null string as their initial value.

Aggregate values come in two types: arrays and tables. Each type is created by a built-in procedure with the same name. Thus:

```
a = ARRAY (20)
```

creates a 20-element array, initializes each element to the null string, and assigns it to a. The second argument to the array procedure is an initializing value:

```
a = ARRAY (20, -1)
```

makes a an array of 20 elements, each with value —1. To get multi-dimensional arrays, express the dimensions as a string:

```
a = ARRAY ('4,5', -1)
```

One can also give explicit lower bounds:

```
a = ARRAY ('-10:10')
```

The default lower bound is 1.

Array elements are referenced by Algol-like subscripts:

```
a[i,j] = a[i,j] + b[i,k] * c[k,j]
```

Each array element behaves as a variable, and can therefore have a value of any data type, independently of any other element.

A table is like a one-dimensional array whose subscripts are not restricted to integers:

```
t = TABLE (20)
```

The argument to the TABLE procedure is an estimate of the maximum number of elements that will actually be stored in the table. If substantially more elements than this are stored, access will begin to slow down. On the other hand, giving too large an initial value wastes space. Once a table has been created, any value can be used for a subscript. `t[a]` and `t[b]` will refer to the same element if and only if a and b are identical. The precise meaning of "identical" is given with the description of the `::` and `:!:` operators; suffice it to say that `3` and `"3"` are not identical, and that after executing

```
a = b
```

a and b are identical regardless of what values they had before.

Binary Operators

The following operators are grouped in order of decreasing priority. Unless otherwise stated, they are left-associative.

`. $` Pattern value assignment (see Patterns)

`^` Exponentiation (right-associative). The right argument must be an integer. If both arguments are integers, the right argument must be non-negative

`* / %`
Multiplication, division, and remainder. As in C, and as not in SNOBOL4, multiplication and division have the same precedence.

`+ -` Addition and subtraction.

`== != < > <= >= :==: :!=: :<: :>: :<=: :>=: :: :!:`
Comparison predicates. Each of these operators returns a null string if the indicated relation holds, and fails if not. The first six do numeric comparisons: an error results if either operand cannot be converted to a number. The next six do string comparisons: an error results if either operand cannot be converted to a string. The last two test if the two operands are identical. Values of different data types are never identical. Strings and numbers are identical if their data types and values match. Other values are identical if they refer to the same object.

`&&` Concatenation. The left operand is evaluated first; if its evaluation fails, the && operator fails. Otherwise, the right operand is evaluated, and && fails if the right operand fails. If either operand is the null string, the result is the other operand, even if that operand is not a string. Otherwise, both operands are converted to strings and the result is their concatenation. An error results if either operand cannot be converted to a string. Note that if the operands of && are predicates, && can be used as a kind of logical conjunction.

`||` Logical disjunction. The left operand is evaluated first; if it succeeds, its value is the value of `||`. Otherwise, the value is that of the right operand. If both operands fail, `||` fails.

`|` Pattern alternation. See *pattern matching* for details.

= Assignment. This operator is right-associative.

? Pattern match operator. The left operand is converted to a string and searched for the first substring that matches the pattern given by the right operand. If no such substring is found, the operator fails. If the left operand is a variable, ? may be used on the left side of an assignment.

Unary Operators

Unary operators bind more tightly than all binary operators.

+ - Unary plus and minus. The result is always numeric, so unary plus is sometimes used for type conversion. Because unary operators bind so tightly, expressions that look like negative constants behave that way for all practical purposes.

. Name operator. The operand must be a variable; the result is essentially a pointer to the variable, similarly to the unary & operator in C. The result of applying the DATATYPE function to the result of the . operator is the string "NAME".

$ Indirection. The operand is converted to a name; the result is the object thus named. $ always yields an lvalue.

? Query. If its operand fails, ? fails. If its operand succeeds, ? yields a null string. Useful for evaluating an expression solely for its side effects.

~ Logical negation. If its operand fails, ~ yields a null string. If the operand succeeds, ~ fails.

& Keyword value. The operand must be the name of one of a restricted set of variables. The lvalue result is a system variable, whose value affects the execution of the program in some way. System variables are discussed separately.

@ Pattern cursor assignment. See *pattern matching* for details.

* Deferred evaluation. Returns a value of type EXPRESSION that contains all the information necessary to evaluate the operand. The operand is not actually evaluated at this time, but can be evaluated later, either by the EVAL procedure or implicitly during pattern matching.

Statements

Elements in brackets are optional. If the description of a statement is split over more than one line, the statement itself may be split analogously. Snocone does not have a null statement.

expression

The expression is evaluated for its side effects. The result, if any, is discarded.

```
if (expression)
        statement1
[ else
        statement2 ]
```

The parenthesized *expression* is evaluated. If it succeeds, *statement1* is executed, otherwise *statement2* is executed.

```
while (expression)
        statement
```

Behaves similarly to C: the *expression* is evaluated, and if it succeeds, the *statement* is executed and control passes back to the beginning of the `while` statement. Unlike C, Snocone has no *break* or *continue* statements.

```
do
        statement
while (expression)
```

Behaves similarly to C: the *statement* is executed, then the *expression* is evaluated, and if the *expression* succeeds, control passes back to the beginning of the `do` statement.

```
for (expression1, expression2, expression3)
        statement
```

Equivalent to:

```
        expression1
        while (expression3) {
                statement
                expression2
        }
```

```
{
        statement list
}
```

The statements in the list, which may contain zero or more statements, are executed in sequence.

label: *statement*

All labels are global (because all SNOBOL4 labels are global), even across procedure boundaries, so they must be chosen with care. A useful convention is to begin a label inside a procedure body with the name of the procedure and an underscore.

`go to` *label*

The space between `go` and `to` is optional. It is wise not to jump from a point inside one procedure into another. Program execution may be terminated by jumping to the reserved label `END`. Labels `RETURN`, `FRETURN`, `NRETURN`, `ABORT`, and `CONTINUE` are also reserved.

`return [ expression ]`

The current procedure returns to its caller. If the *expression* is given, the procedure yields that value. If no *expression* is given, the value returned is that of the variable with the same name as the procedure; if that variable was not assigned in the procedure, the null string is returned.

`freturn`

The current procedure returns and fails.

`nreturn [ expression ]`

The current procedure returns `$(`*expression*`)` as an lvalue. If the *expression* is not given, the indirection is applied to the variable with the same name as the procedure. An error results if this variable was not given a value inside the procedure.

Procedures

Here is an example of a procedure declaration:

```
procedure gcd (m, n) {
        while (m != n) {
                if (m > n)
                        m = m % n
                else
                        n = n % m
        }
        return m
}
```

Arguments are passed by value, but note that the value passed for an aggregate argument (array, table, or structure) is really a pointer to the aggregate itself.

If a procedure is called with too few arguments, extra null strings are supplied as necessary. If called with too many arguments, the extras are quietly ignored.

Local variables can be nominated for a procedure:

```
procedure f(x) y, z {...
```

All variables are global, but name scoping is dynamic. One way to look at it is to imagine that when a procedure is entered, all the variables defined in that procedure are saved and set to null. Those variables are then restored when the procedure returns.

Thus, the following example prints 5 and then 1:

```
a = 1
f()
g()

procedure f() a {
        a = 5
        g()
}

procedure g() {
        OUTPUT = a
}
```

The following example also prints 5 and then 1:

```
a = 1
b = .a
f()
g()

procedure f() a {
        a = 5
        g()
}

procedure g() {
        OUTPUT = $b
}
```

Local variables can only be associated with procedures. Procedures are recursive.

Input-Output

All I/O is done through "associated variables". A variable may be input-associated or output-associated (or both). Whenever a value is assigned to an output-associated variable, that value is automatically written in the file associated with that variable. Whenever a value is requested for an input-associated variable, a line is read from the file associated with that variable, and the contents of the line are used for the value. When the end of an input file is reached, any attempts to access variables associated with that file will fail.

Initially, the variable `OUTPUT` is output-associated with the standard output file, the variable `INPUT` is input-associated with the standard input file, and the variable `TERMINAL` is both input- and output-associated with the user's terminal.

Thus, the following program copies its standard input to its standard output, a line at a time:

```
while (OUTPUT = INPUT) {}
```

New associations are formed by the `INPUT` and `OUTPUT` procedures:

`INPUT` (*name, channel, file*)
`OUTPUT` (*name, channel, file*)

In both cases, *name* is the name of the variable to be associated (the name of the variable *x* is `.x`), and *file* is the file to be used. *Channel* is a string that you will use to identify subsequent operations on that file. Internally, there is a one-to-one correspondence between channel names and file descriptors.

The file argument must not be given for other than the first call to INPUT or OUTPUT on a given channel, as a channel can only be connected to a single file.

Other procedures dealing with input-output are:

SET (*channel, offset, whence*)

This function repositions the file specified by its first argument in a system-dependent manner. In implementations running under the UNIX system, the behavior is similar to the *lseek* system call.

REWIND (*channel*)

Repositions the named channel to the beginning of the file.

ENDFILE (*channel*)

Indicates that you are done using the given channel. All variables associated through that channel are disassociated, output buffers are flushed (if any), and the file is closed.

DETACH (*name*)

Disassociates the named variable. Does not close the file: a later call to INPUT or OUTPUT can reassociate it.

Pattern Matching

A *pattern* is a data structure that describes a class of strings. Patterns are used by the ? operator, which determines if the string given as its left operand contains a substring described by the pattern given as its right operand. The part of the Snocone system that does this is called the *scanner*. The input to the scanner is the string to be searched, called the *subject*, and the pattern sought.

The scanner tries to find a substring of the subject that is matched by the pattern. It first looks at substrings starting at the first character of the subject. If it doesn't find one, it tries the ones starting at the second subject character, and so on. If the scanner cannot find an appropriate substring, the pattern match fails.

If the &ANCHOR system variable is nonzero, the scanner only looks at substrings that start at the first character of the subject. This is said to be an *anchored* pattern match. The &ANCHOR variable is zero at the start of program execution.

The important part of pattern matching is therefore determining whether the subject contains a substring that starts at a given character and matches a given pattern. To understand how this is done, we must take a closer look at patterns.

Every pattern is a concatenation of one or more *elements*, each of which has zero or more *alternatives*. Each alternative may itself be an arbitrarily complicated pattern. Some patterns have alternatives that are determined dynamically as they are needed during pattern matching.

If a pattern has only one element, the scanner determines if it matches at a given point by trying to match each of the pattern's alternatives at that point. If no alternative matches at that point, the element cannot be matched.

If the pattern has more than one element, matching that pattern at a given point means: (a) finding an alternative for the first element of the pattern that matches a subject substring starting at the given point, and that also (b) allows the remaining elements of the pattern to match a substring that starts immediately after the substring matched in (a).

During pattern matching, the scanner keeps its place by means of an internal value called the *cursor*, which represents the number of characters in the subject that precede the current location. These characters are counted from the beginning of the subject even when the scanner is trying to match a substring that starts at some other place in the subject.

The concatenation of two patterns P1 and P2 is a pattern whose elements are P1 and P2. The alternation operator ¦ similarly constructs alternatives. When a string is used as a pattern, it matches only itself. Thus,

```
"a" ¦ "e" ¦ "i" ¦ "o" ¦ "u"
```

is a pattern that matches any lower-case vowel.

Consider the following statement:

```
P1 = ("ab" ¦ "a") && ("b" ¦ "c")
```

This assigns to P a pattern with two elements. The first one matches either ab or a, and the second matches either b or c. Look at:

```
"ab" ? P1
```

The scanner first tries the first alternative of the first element of P1 by matching ab in the subject with ab in the pattern. This alternative matches, so the element ("ab"¦"a") matches, and the scanner goes on to the second element ("b"¦"c"). Here, it will be unsuccessful in matching both b and c, because it has already exhausted all the characters of the subject. Thus the scanner fails to match the second element of P1, and must back up and rematch the first element. Fortunately, the first element has an alternative in a; this matches, and now the scanner can try to match the second element ("b"¦"c") again. The first alternative (b) succeeds, so the ? operator therefore succeeds. If we wanted to examine just how the pattern elements matched, we could have written:

```
P1a = ("ab" ¦ "a") . OUTPUT && ("b" ¦ "c") . OUTPUT
"ab" ? P1a
```

This would print a and b on separate lines, showing that "ab"¦"a" matched a and "b"¦"c" matched b.

When any pattern is first encountered during pattern matching, it will match some string, and when the scanner backs into it, it may match some other string. Because many patterns behave differently on their initial match than when rematched, it is useful to describe the two cases separately.

For instance, when a string is used as a pattern, it initially matches itself. If the scanner later backs into it, it fails.

As another example, consider P1¦P2. This pattern matches every possibility for P1, and when it has exhausted P1, then matches every possibility for P2 before finally failing.

With this in mind, we can describe the various pattern-matching operators, procedures, and pre-defined variables:

`P1 && P2` Tries to match P1, fails if it can't. Then tries to match P2. If P2 fails, tries the next alternative for P1, and then tries P2 again. Eventually, it either runs out of alternatives for P1, in which case P1&&P2 fails, or it finds alternatives for both P1 and P2 which allow them to match.

P1 ¦ P2 — Tries to match P1. If successful, P1¦P2 matches the string that was matched by P1. Otherwise, matches whatever P2 matches, and fails if P2 fails.

P $ V — Tries to match P. If P matches, a copy of the substring matched by P is immediately assigned to the variable V.

P . V — Tries to match P. If the entire pattern match of which this element is a part is ultimately successful, a copy of the substring matched by P is assigned to V after the entire match completes.

@N — Matches a null string, and immediately assigns cursor position to the variable N. The cursor position is the number of characters that precede this null string in the subject of the entire pattern match. Thus:

```
"abcde" ? @OUTPUT && "c"
```

prints 0, 1, and 2 on separate lines.

ABORT — The entire pattern match is aborted.

ARB — A pre-defined variable that matches anything at all. More specifically, it initially matches the null string. When backed into, it matches a string one character longer than the one it matched last time.

BAL — A pre-defined variable that matches a non-empty parenthesis- balanced string. This string is balanced only with respect to parentheses, and not any other kinds of brackets.

FAIL — A pre-defined variable that always fails to match. It is sometimes useful to force the scanner to try all alternatives for a pattern. For instance:

```
s ARB $ OUTPUT && FAIL
```

prints all substrings of s.

FENCE — Initially matches the null string, but if the scanner backs into it, the entire pattern match is aborted. Identical to ""¦ABORT.

REM — Matches from the current position to the end of the subject. Identical to RTAB(0).

SUCCEED — Identical to ARBNO("")

ANY(s) — Matches any single character in s.

ARBNO(p) — A pre-defined procedure that yields a pattern that matches zero or more copies of the pattern p. Initially matches the null string. Each time the scanner backs into it, it tries to extend the substring already matched by matching one more instance of p.

BREAK(s) — Matches a string starting at the current position up to but not including the next character in the subject that is also found somewhere in s. Failure if no such character is found. No alternatives.

BREAKX(s) — Like BREAK, but if backed into, it matches up to the next subject character also found in s, and so on.

LEN(n) — Matches exactly n characters.

NOTANY(s) — Matches any single character not in s.

POS(n)	If exactly n subject characters precede the current position, POS matches the null string. Otherwise it fails.
RPOS(n)	If exactly n subject characters remain to be matched, RPOS matches the null string. Otherwise it fails.
RTAB(n)	If at least n subject characters remain in the subject, RTAB matches characters until exactly n remain. Otherwise it fails.
SPAN(s)	Starting at the current position, matches as many characters as possible taken from s.
TAB(n)	TAB matches from the current position forward up to and including the nth character of the subject. If more than n characters have already been matched in the subject string, TAB fails.

All pattern-valued procedures can take an unevaluated expression as argument. The expression will be evaluated when the pattern element is matched.

System Variables

The & operator looks at the name of its operand, not its value. For each of a small set of names, & yields a *system variable*, whose value affects the operation of the system in some way. For instance, we have already seen &ANCHOR, which controls whether or not the scanner is restricted to examining initial substrings of the subject. The complete list of system variables is:

&ABORT	The same value as the pre-defined variable ABORT.
&ALPHABET	A string that contains all the characters of the machine's collating sequence, in order.
&ANCHOR	If zero, all pattern matches are unanchored, otherwise they are anchored.
&ARB	The same value as the pre-defined variable ARB.
&BAL	The same value as the pre-defined variable BAL.
&CODE	This variable is initially zero. Its value is returned to the operating system when the program finishes executing.
&DUMP	This variable is initially zero. If it is 1 at the end of execution, the values of all variables are printed. If it is 2, the values of all array, table, and structure elements are also printed.
&FAIL	The same value as the pre-defined variable FAIL.
&FENCE	The same value as the pre-defined variable FENCE.
&FNCLEVEL	The current level of procedure nesting.
&INPUT	If this variable is set to 0, all input association is suspended.
&MAXLNGTH	The maximum length of a string. This value cannot be increased beyond its initial value.
&OUTPUT	If this variable is set to 0, all output is suppressed until it is again set nonzero.
&REM	The same value as the pre-defined variable REM.
&STCOUNT	A count of how many statements have been executed so far. Because this counts SNOBOL4 statements, not Snocone statements, it should be considered only approximate.

&STLIMIT	When &STCOUNT reaches this value, execution terminates with an error message. It is initially 50000, and may be set freely. If it is set to a negative value, statement counting is disabled.
&SUCCEED	The same value as the pre-defined variable SUCCEED.

Examples

Hello World

This is the canonical sample program:

```
OUTPUT = "Hello world!"
```

The present implementation of Snocone is case-insensitive in identifiers, but future versions may well become case-sensitive. If so, it is likely that pre-defined names, such as INPUT and OUTPUT, will have to be written in upper case. Keywords, such as if and while, will be written in lower case.

Topological Sorting

We now develop a topological sorting program based on the algorithm described on page 262 of *Fundamental Algorithms*12 The reader may wish to compare this program with the SNOBOL4 program based on the same algorithm that appears on pages 221-222 of *The SNOBOL4 Programming Language*.

The input is a set of pairs of objects, where the first object in each pair is considered to precede the second. The output is a list of objects in a sequence that meets all the constraints implied by the input. In other words, the program generates a total ordering that includes a given partial ordering.

If the input contains a loop, the program will detect this fact and complain.

For example, given the following input:

```
letters alphanum
numbers alphanum
blanks optblanks
numbers real
numbers integer
letters variable
alphanum variable
binary binaryop
blanks binaryop
unqalphabet dliteral
unqalphabet sliteral
sliteral literal
dliteral literal
integer literal
real literal
```

the program will produce the following output:

```
letters
numbers
blanks
binary
unqalphabet
alphanum
real
integer
optblanks
binaryop
dliteral
sliteral
variable
literal
```

The basic strategy of the program is simple: for each object, we remember how many immediate predecessors it has, and store a list of all its immediate successors. When we have finished reading the input, we can immediately output the objects that have no predecessors. Each time we output an object, we remove it from the data structure and decrement the predecessor count of each of its immediate successors.

We will eventually reach a state in which we run out of objects without predecessors. When that happens, we are done. If any objects remain, they form a loop.

To reduce the time spent searching for objects without predecessors, we keep a queue of such objects. We must also keep a queue of the successors of each object. In both cases, we could use a stack instead of a queue, but using a queue tends to favor the order in which objects appeared in the input, which makes the output more intuitively useful.

The following structure declarations and subroutines manipulate queues. A queue consists of a header (of type queue) which points to the first and last elements of a singly-linked list of queue elements (of type qel).

```
struct queue {head, tail}
struct qel {obj, link}

procedure enqueue (q, x) y {
        if (head(q) :: "")
                head(q) = tail(q) = qel(x)
        else {
                y = qel(x)
                link(tail(q)) = y
                tail(q) = y
        }
}

procedure dequeue (q) x {
        if (head(q) :: "")
                freturn
        x = head(q)
        if ((head(q) = link(x)) :: "")
                tail(q) = ""
        return obj(x)
}
```

By convention, we use the null string to indicate the end of a list. This is convenient because uninitialized variables and structure fields and missing arguments are automatically set to the null string. Thus, the test

```
head(q) :: ""
```

is a convenient way of testing whether head(q) has been set or not.

We represent each object as a structure containing the object's name (so we can print it), the count of immediate predecessors, and the queue of successors:

```
struct object {name, count, suc}
```

Since we will be reading the names of objects rather than the objects directly, we will need to map names to objects. This can easily be done with a table and a mapping subroutine that creates elements in the table as needed:

```
namemap = TABLE()
objects = queue()

procedure getobj (name) {
        if ((getobj = namemap[name]) :: "") {
                getobj = namemap[name] = object (name, 0, queue())
                enqueue (objects, getobj)
                nobj = nobj + 1
        }
}
```

This procedure uses the feature that if no return value is explicitly given, the value of the variable with the same name as the procedure is used. If an appropriate table element already exists, the :: operator will fail and the value of getobj will be the value retrieved from the table. As a side effect, we maintain a global queue of all known objects in objects and count them in nobj.

Now that we can map from names to objects, it is an easy matter to enter a new relation into our data structure. Procedure enter takes the names of two objects:

```
procedure enter (p, q) {
        p = getobj (p)
        q = getobj (q)
        count(q) = count(q) + 1
        enqueue (suc(p), q)
}
```

We first locate the objects to which p and q refer, creating them if necessary. Since p precedes q, we increment the predecessor count of q and append q to the successor list of p.

Building the data structure is now just a matter of scanning the input file:

```
while (line = INPUT) {
        if (line ? FENCE && BREAK(' ').p && SPAN(' ') && rem.q)
                enter (p, q)
        else
                TERMINAL = "bad input line: " && line
}
```

The pattern match assumes that everything up to the first blank in the input line is the first object in a relation, and everything after the first blank is the second object. If the match fails, the program complains.

Once all the input has been read, we must initialize the queue of minimal objects (objects without predecessors). This was the reason for keeping a queue of all objects, which is now destroyed to build the queue of minimal objects. It is not necessary to destroy the queue, but it is more convenient because it is then possible to use dequeue:

```
zeroes = queue()

while (x = dequeue (objects))
        if (count (x) == 0)
                enqueue (zeroes, x)
```

As long as there is a minimal object, we can print its name, delete it, and decrement the predecessor count of each of its successors. If we decrement a predecessor count to zero, that object is now minimal.

```
while (x = dequeue (zeroes)) {
        nobj = nobj - 1
        OUTPUT = name(x)
        while (y = dequeue (suc (x))) {
                if ((count(y) = count(y) - 1) == 0)
                        enqueue (zeroes, y)
        }
}
```

This loop runs until there are no more minimal objects. If there are still elements remaining (nobj is nonzero), then those elements form a loop.

```
if (nobj != 0)
        TERMINAL = "The ordering contains a loop."
```

References

1. R. E. Griswold, J. F. Poage, and I. P. Polonsky, *The SNOBOL4 Programming Language (second edition)*, Prentice-Hall, Englewood Cliffs, New Jersey (1971).

2. R. E. Griswold, *String and List Processing in SNOBOL4; Techniques and Applications*, Prentice-Hall, Englewood Cliffs, New Jersey (1975).

3. J. F. Gimpel, *Algorithms in SNOBOL4*, John Wiley and Sons, New York, New York (1976).

4. B. W. Kernighan, "RATFOR — A Rational Fortran," *Workshop on Fortran Preprocessors, Pasadena, Calif.*, p.3 (November 1974).

5. Stuart I. Feldman, "The Programming Language EFL," Comp. Sci. Tech. Rep. No. 78, Bell Laboratories, Murray Hill, NJ (June 1979).

6. David R. Hanson, "RATSNO — An Experiment in Software Adaptability," *Software — Practice and Experience* 7, pp.625-630 (1977).

7. R. E. Griswold, "Rebus — a SNOBOL4/Icon Hybrid," Technical report TR 84-9, Department of Computer Science, Tucson, Arizona.

8. R. E. Griswold, *The Icon Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1983).

9. Robert B. K. Dewar and A. P. McCann, "Macro Spitbol — a SNOBOL4 Compiler," *Software — Practice and Experience* 7, pp.95-113 (1977).

10. S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," *Bell Sys. Tech. J.* 57(6), pp.1971-1990 (1978).

11. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell Sys. Tech. J.* 57(6), pp.1905-1929 (1978). Also in *Comm. ACM*, 17, pp. 365-375 (1974).

12. Donald E. Knuth, *Fundamental Algorithms*, Addison-Wesley (1973). Volume 1 of *The Art of Computer Programming*

Camphor: A Programming Environment for Extensible Systems

Michael Leon Kazar

Information Technology Center
Carnegie-Mellon University
Pittsburgh PA 15213

1. Introduction

UNIX† contains many examples of programs that have succeeded because their users have been able to extend them procedurally; *troff* and *EMACS* are two obvious examples. In each case ad-hoc techniques such as interpreters for special languages were used, primarily because the UNIX C environment insists on linking all the code into an *a.out* file before it is executed.

The Camphor environment provides for parts of an application to be written in the form of modules that are linked into a process during its execution. It consists of a compiler that generates pure, position-independent code, and a run-time system that traps references to missing modules, searches a path for them, and links them into the running process. By placing private modules in this path, users can customize and extend applications to satisfy their own needs.

Camphor's dynamic linking has another benefit. Applications using many conventional libraries must be re-linked whenever any one changes. Ensuring that they are up-to-date in a large distributed environment is time-consuming. But applications dynamically linking modules acquire up-to-date versions whenever they are executed.

2. Previous Work

User extensibility can be supported in at least one of three broad schemes:

a) By including in the application to be extended an interpreter for a suitable language. EMACS has successfully exploited this: its original [Stallman 81] incarnation was written in interpreted TECO and could be extended in the same language, while UNIX EMACS [Gosling 83] included an interpreter for MockLisp, a simple language designed for the purpose.

b) By link-editing .o files on demand at run-time. Franz lisp provides this facility.

c) By compiling the extensions into position-independent code, which then needs no link-editing. It can simply be read in, (or mapped in, on some systems) wherever convenient. Multics [Organick 72], for example, maps in its object code.

Camphor uses the last technique. Extensions are written in a special programming language whose compiler generates pure, position-independent code. The first reference to an external routine is trapped, the routine loaded into the heap, and the reference completed.

3. The Language

Camphor programs are organized into **modules**. A module is a set of procedures and static variables, dynamically loaded as a unit from a single file.

Syntactically, the language resembles Pascal, with some of the syntactic sugar of Xerox's Mesa. Data types include short and long integers, characters, pointers to elements, arrays of elements, and records of elements. Control constructs include if-then-else, begin-end blocks, while loops, and constructs for exiting from a loop. There is no unrestricted goto statement.

† UNIX is a Trademark of Bell Laboratories.

3.1 Calling Procedures

Calls in Camphor are made to a procedure within some module. Procedures are named by strings having two components separated by "$":

module-name$procedure-name

In a language such as C that provides only static linking, procedure names can occur in only one context: a procedure declaration.

Camphor procedures, on the other hand, can be named in two contexts.

a) A Camphor procedure can be named directly in a procedure declaration. The name is a compile time constant that is resolved into the address of a procedure at run-time.

b) The Camphor procedure *link$create* can be invoked at run-time with a character string computed at run-time. The procedure *link$create* is called with the string name of a procedure one wishes to invoke, and effectively returns the address of the procedure whose name was passed to *link$create*. The details of this operation are given in the section on the run-time system.

The first method of naming Camphor procedures is by far the most common, and most closely corresponds to the C method of procedure naming. The main difference is simply that with Camphor, the name is interpreted at run-time, while with C, it is interpreted at link-time. However, both C and Camphor specify the name at compile-time.

The second Camphor naming method is the most interesting, since it allows a program to generate procedure names at execution time. This flexibility is important for some of our applications; this point will be described in detail later on in this paper.

3.2 Mixing Camphor and C Procedures

Camphor uses the same calling convention as the C compiler. As a result, given a pointer to a procedure, any program, whether written in Camphor or in C, can call any other procedure. Of course, C procedures must be present in the process image at the start, since the Camphor run-time can not load C produced *.o* files.

Because the calling conventions are identical, the most interesting issue in these cross-language procedure calls is how a procedure written in one language specifies the name of the procedure in the other language.

An application written in C can invoke routines written in Camphor in two ways:

a) It can invoke them via the dynamic linking mechanism, by calling a library routine:

camphorcall("foo$bar", arg1, arg2);

calls the routine *bar* in module *foo* with arguments *arg1* and *arg2*, loading the module if necessary.

b) Once loaded, a Camphor routine can store pointers to itself, or other Camphor routines, in data structures. C programs can call indirectly via these pointers in the normal way.

Similarly, Camphor routines can also call C procedures. Camphor procedures can name C procedures in two ways:

a) Camphor can call indirectly via pointers to functions stored in C data structures.

b) When an application written in C initializes the Camphor system, it may *export* certain of its routines to the Camphor programmer by including pointers to them in a data structure passed as an argument to **camphorinit()**. These routines form a pseudo-module called **sys**. Thus, the C library's **malloc()** might be invoked from Camphor as *sys$malloc()*.

4. The Compiler

The compiler consists of four main phases. The first phase is the parser, implemented with Lex and Yacc. The parser feeds an intermediate code generator that generates triples (sometimes referred to as two-address code); these triples are then optimized and used for code generation. The parser and the intermediate code generator are all completely machine independent.

The next phase of the compiler applies machine independent optimizations to the triples generated by the intermediate code generator. The optimizer divides the triples into basic blocks, eliminates unreached blocks, and performs constant folding. It also performs some live/dead variable analysis, trying to identify values used by subsequent expressions that might profitably be kept in registers. The only machine-dependent parameter used by the optimizer is the number of registers available. While a machine independent optimizer aids in porting the compiler, it also means that the optimizer cannot know exactly which expressions, the machine-dependent code generator can actually utilize.

The code generator contains all the machine dependent parts of the compiler, although much of its work is actually performed by a machine independent subroutine library provided by the optimization module. In particular, the optimization module provides a set of routines to keep track of the machine state, most importantly, the contents of the machines registers at all points in the program being compiled.

The contents of a register are represented by a set of expression triples. The presence of a triple in such a register's set means that the expression can be found in that register. Procedures provided by the optimization module allow the caller to

a) determine whether a particular triple can be found in a register.

b) specify that any expression depending on a particular variable is no longer valid. This is called **killing** an expression.

c) kill any expression dependent upon values of a particular storage class, such as statics or heap-allocated objects, which are generally assumed to be killed by a procedure call.

d) kill any expression of a particular type. This is done when a pointer to an object of a particular type is de-referenced on the left-hand side of an assignment statement.

e) specify that a particular triple can now be found in a particular register.

f) specify that a particular register no longer contains any useful triples.

Through the use of these optimization library procedures, a code generator can painlessly keep track of the contents of the machine's registers throughout the compilation process.

The code generator produces code for each basic block in the order in which the blocks in the source program occur. Ignoring loops for the moment, when the code generator processes a basic block, it has already generated code for all the basic blocks that can transfer control to the current block. The code generator starts off a basic block by calling optimization procedures that merge the possible machine states resulting from all of the basic blocks immediately preceding the current block. Starting with the block's initial machine state, the code generator processes each triple in the block, making use of the current contents of the machine's registers, and continually updating the machine's register state after the code for each triple is generated.

When the start of a loop is encountered, the code generator throws away all knowledge of the machine's state. In the future, the machine independent optimizer may be modified to better handle loops.

Note that the compiler directly generates executable code for the target machine. There is no assembly language pass. Code generators exist for both the VAX and the 68000.

5. The Run-Time Support

The run-time system is responsible for executing calls to dynamically-linked procedures as rapidly as possible.

The compiler represents a call to an external procedure (e.g. *foo$bar*) as a call to a static structure, called a **link**, containing approximately ten bytes of illegal instructions, and a string giving the name of the called procedure, in this case *foo$bar*. On the first call to the procedure, a trap is generated, invoking the Camphor run-time system. It checks to see if module *foo* is already loaded. If not, the directories in the **BZPATH** environment variable are searched for a file named *foo*. If found, the entire file is read into space allocated for the purpose.

After the module is loaded, the run-time system still must locate the desired procedure within the module. Each Camphor module contains a symbol table, which, among other things, gives the offset within the module of each external procedure in the module.

Camphor object files are designed to interact well with mapped files, should they ever be provided by UNIX. Code generated by the Camphor compiler is position-independent. References to local variables are indexed by the stack pointer, as in C. References to static variables are indexed by another register, initialized by the link used to call the procedure, This allows the Camphor run-time system to allocate a module's *static* variables independently from the module's code. Thus code can be mapped into a process, and shared, with each process having a separate copy of its *static* variables.

This sharing will be important to the Information Technology Center (the ITC). We have many fairly small applications making use of very large subroutine libraries. Since UNIX only shares code when the two processes' text regions are identical, two different applications sharing the same libraries actually share no code. Sharing Camphor libraries should greatly reduce the memory requirements of our system.

Once the run-time system has loaded the module, initialized the module's *static* variables, and located the target procedure, the target procedure can be invoked.

Unfortunately, doing all this work on every procedure call would dramatically increase the procedure call overhead. To get good average-time performance for calls, the run-time system also replaces the illegal instruction at the head of the link with a pair of instructions. The first instruction loads a register with a pointer to the writable copy of the target procedure's *static* variables. The second instruction branches to the true location of the code in this process. The overhead on subsequent calls through this link is just these two instructions.

Aside from actually resolving dynamic links upon their first use, the Camphor run-time system exports many of the same functions used internally to resolve links to Camphor programs. The most important of these procedures is *link$create*, one use of which is described in the next two sections.

6. Using Camphor

The ITC has used Camphor to extend two systems, the Base Editor subroutine library, and the window manager.

Many applications at the ITC use the Base Editor library developed originally by James Gosling. This provides a library of data types bundled with their visible behaviour, including buttons, scroll bars, and panes of multi-font dynamically re-formatted text. We would like to extend this editor with many other data types, such as drawings, graphs, animated figures, and to allow users to define their own types. Dynamic linking allows individual users to extend this editor themselves; their new sub-editors are loaded into the running editor when they are first referenced. A version of the Base Editor implemented almost entirely in Camphor is being tested. Camphor is used in this editor in two ways.

First, the editor defines a standard procedural interface to any type of editable object; new types of objects can implement their behaviour through a module providing these procedures. When a Base Editor process encounters an instance of a new object type in a document, for example a *piechart*, it can dynamically link in the *piechart* module and call its standard procedures to draw the object, handle mouse clicks on the object, and so on. In this way anyone can define a new type of object and implement it as a camphor module. From that point on, all programs using the Base Editor subroutine library for their editing can use these new objects.

The editor communicates with a sub-editor handling a new type of object by means of objects called **layouts**. A layout object contains two important components,

a) a pointer to the object being edited and

b) a set of pointers to procedures implementing useful operations on this object.

The implementation procedures are type-specific, that is, a different set of procedures are provided for each new type of editable object. Five of the procedures must be provided by every editable object. These are

a) **Update**. This procedure ensures the state of an object matches its displayed representation on the screen.

b) **FullUpdate**. This procedure performs the same function as *Update*, but assumes that the screen has just been cleared.

c) **SetDimensions**. The editor uses *SetDimensions* to tell a sub-editor how much screen space the sub-editor may use for this object.

d) **GetDimensions**. This procedure returns the amount of the screen space offered by *SetDimensions* that the sub-editor actually requires.

e) **MouseEvent**. This procedure is called by the editor when a mouse click occurs in the region of the screen containing this layout.

f) **Read**. This procedure creates a new object of an abstract type.

Several other layout procedures are optional, and are called only if they are provided.

The abstract objects placed in layouts have names, and for each type of object, a module with the same name provides the procedures used by the layout package to communicate with the type-specific editor.

Consider the example of a new type of editable object, a *piechart*. The layout package assumes that five procedures are provided by the *piechart* module: *piechart$Update, piechart$FullUpdate, piechart$GetDimensions, piechart$SetDimensions, piechart$MouseEvent* and *piechart$Read*. At execution time, when the editor first encounters an object of type *piechart,* it calls *link$create* to generate references to these five procedures. The editor also attempts to locate the optional procedures in the *piechart* module. Thus dynamic linking allows the editor to locate precisely those sub-editors required by a particular document. The document itself contains only the name of the abstract type, not the actual code. The actual code is loaded at its first reference.

The second important use of dynamic linking occurs during the editor's initialization. At this time, the editor invokes the procedure *main* from the module *bxpro,* if it exists. This user-defined function can, for example, define new commands to the editor and bind them to keys. This use of Camphor in extending the editor is quite similar to the use of MockLisp in extending Emacs.

Dynamic linking interacts strongly with the software release process. Eventually, there may well be hundreds of applications using the Base Editor subroutine library. Many of these applications will also involve user-defined layouts (called **insets**), and will thus need to include the code implementing these layout types. Yet an application program may, while procesing documents produced by another application, encounter insets that are functionally a part of the first

application. Furthermore, system programs such as the mail reading program must handle documents containing any of these objects. Now, if every release of the system required collecting all of the users' layout modules, adding them to the main library, and then re-linking all of the Base Editor applications, the release process would be a nightmare. However, using dynamic linking, simply adding new layout modules to an appropriate search path ensures that all Base Editor applications handle objects of the new type.

7. Other Camphor Applications

The ITC's window manager runs as a user-level server process to which client applications make remote procedure calls (RPC) in order to perform line drawing and pixel manipulations. The RPC mechanism is built on top of 4.2 BSD UNIX sockets, and thus incurs a 20 millisecond round trip overhead. Despite the RPC overhead, the performance is generally acceptable, but some tasks remain difficult:

a) Some applications require particular output primitives (e.g. spline curves). Approximating them by short lines consumes too much socket bandwidth.

b) Some interactive techniques (e.g rubber-band lines) require very rapid feedback. The RPC and scheduling overhead of implementing them in the client is prohibitive.

By providing clients with the ability to cause the window manager to invoke arbitrary Camphor procedures, we allow applications to define their own output primitives. By incorporating a few hooks in the window manager, such as a pointer to a routine that is called every time the mouse moves in a window, we allow applications to define their own styles of interaction. These capabilities are similar to the ability to down-load programs into a window on the Bell Laboratories' Blit.

8. Acknowledgements

Many of the ideas in this paper derive from the implementation of dynamic linking in the Multics system [Organick 72].

Thanks go to David Nichols, who implemented the Camphor parser, and assisted in the development of the Camphor Base Editor. David Rosenthal modified the ITC window manager to work with Camphor. Both David Nichols and David Rosenthal provided me with many helpful comments concerning this paper. David Dill, David Nichols and Barak "David Bigboote" Pearlmutter all participated in many interesting discussions concerning type systems.

Finally, institutional thanks go to both the International Business Machines corporation and Carnegie-Mellon University for providing a fine environment in which to work.

9. References

[Dahl 1972] O. J. Dahl and C. A. R. Hoare, *Structured Programming,* pp 175-220, Academic Press, New York, New York (1972)

[Gosling 1983] J. A. Gosling, *UNIX Emacs,* Carnegie-Mellon University Computer Science Department, Pittsburgh, Pennsylvania (1983)

[Organick 1972] E. I. Organick, *The Multics System: An Examination of its Structure,* MIT Press, Cambridge, Massachusetts (1972)

[Stallman 1981] R. M. Stallman, *EMACS Manual for TOPS-20 Users,* MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts (1981)

A Recipe for Establishing Point-to-Point TCP/IP Network Links with 4.2 BSD UNIX

Thomas Ferrin

Computer Graphics Laboratory
School of Pharmacy
University of California
San Francisco, CA 94143
ucsfcgl!tef

ABSTRACT

The networking capabilities provided with 4.2BSD UNIX along with the recent proliferation of Local Area Networking (LAN) hardware have encouraged many sites to install their own LANs. Most often these LANs are based on either Ethernet or ring network technology and hence are limited in the interconnection distance between hosts. It therefore often becomes desirable to interconnect LANs with dedicated point-to-point network links utilizing a host on each LAN as an internetwork "gateway" machine.

This short paper describes the procedures necessary for establishing your own point-to-point TCP/IP network link. Since these procedures often involve dealing with several hardware vendors as well as common carrier providers, the task of interconnecting networks often appears overly complex. The necessary hardware, software, and costs for establishing either a 9600 baud or a 56K baud dedicated point-to-point link are described, along with the techniques necessary for maintaining such a link. Both common carrier (i.e. telephone company) and private (e.g. microwave link) techniques are detailed, as well as the software changes necessary to 4.2BSD to support such a link. A working network link between UCSF and UCB is used as a model.

Introduction

The advent of recent technical advances and cost reductions in local area network (LAN) technology has encouraged the proliferation of networks at many UNIX† sites. These LAN's are typically based on Ethernet‡ or ring network hardware and often utilize the Department of Defense's Advanced Research Projects Agency (DARPA) TCP/IP communication protocols. In particular, the recent 4.2BSD and 4.3BSD UNIX software distributions from UC Berkeley have incorporated TCP/IP within the operating system.

A common limitation found in local area networks is the maximum separation distance between two hosts on the net. For example, the maximum distance between hosts on an Ethernet based LAN is 1,600 meters (2,800 meters with a remote repeater). Thus, LAN's are typically limited to installations within a single building or complex of buildings such as a university campus or industrial complex. Often this limitation is more restrictive than desirable and one possible solution is to establish multiple LAN's that are interconnected via point-to-point dedicated network links. Figure 1 illustrates such an interconnected network. In such a configuration, data packets must typically be stored and forwarded across network boundaries; we refer to hosts which store and forward packets as "gateways".

Although there are commercial inter-Ethernet links available (e.g. TransLAN by Vitalink), these devices are relatively expensive (~$40,000/pair) and currently not available for ring based architectures. Thus, the remainder of this paper discusses implementing point-to-point

† UNIX is a trademark of AT&T Bell Laboratories.

‡ Ethernet is a trademark of Xerox, Digital Equipment and Intel Corporations.

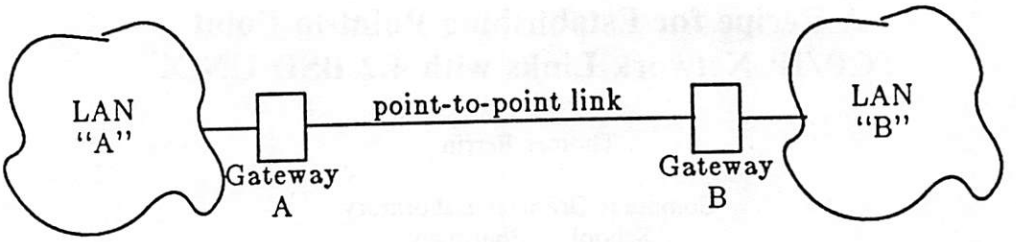


Figure 1. Interconnected Local Area Networks

internetwork links utilizing existing network hosts as gateways.

Parameters and Options

The following discussion is by no means meant to be exhaustive. We confine ourselves to discussing possible approaches to interconnecting networks running 4.2BSD UNIX; although much of the discussion can be applied to other systems, especially the sections on hardware, software details are likely to be different with other operating systems. When discussing hardware we make the following additional disclaimer: the examples given are exactly that -- examples -- and do not necessarily represent either the best performing or lowest cost hardware available. The data communications marketplace is undergoing rapid transformations with both technology and costs changing almost daily. Our experience is therefore limited and we confine ourselves to discussing what we know to work.

Our model consists of a working internetwork link connecting two Ethernet based LAN's separated by a line-of-site distance of 13.3 miles. The LAN's are located on the University of California's San Francisco and Berkeley campuses.

The major parameters that govern data communication decisions when designing point-to-point dedicated network links are: i) distance between gateway hosts and ii) desired communications speed. We loosely group these parameters as follows: distances are categorized as "in house", that is within a building or building complex, "local", usually less than 5 miles of cable between gateways, and "distant" or "long haul" -- anything over 5 miles; for speeds we consider 9.6K bits per second (bps), 56Kbps and 1.5Mbps and higher. While this classification scheme may oversimply decision making with respect to the many choices available in today's telecommunications marketplace, it does encompass the majority of possible interface and circuit combinations and adequately serves the need for the discussion which follows.

In-House Circuits

In house installations typically have the freedom to install whatever cabling is necessary for the particular host interface hardware chosen. Examples include twin coax cables for use with Digital Equipment DMR11-CP interfaces or multiconductor twisted pair cables for parallel interfaces such as DEC's PCL11-B. There is often little constraint nor even increased cost for high speed connections, and hence 1Mbps or higher data rates are not uncommon. An inexpensive low speed configuration consists of simply wiring together an RS232 port from each host and installing the appropriate TCP/IP "line disipline" network driver†. The disadvantage of this low cost approach is the additional host loading factor caused by the interrupt per character nature of the asynchronous device interface and the relatively high overhead of the line disipline routines. An intelligent synchronous communications device such as the DMR11, on the other hand, transfers a block (typically 1024 bytes) of data at a time via direct memory access and only

† A version of the asynchronous TCP/IP line disipline software developed by Rick Adams at the Center for Seismic Studies is being distributed as part of 4.3BSD. It is also available from directly from Rick via seismo!rick.

interrupts the host when the transfer is complete and without errors.

For directly connected host interface equipment that terminates in a RS232 or RS449 circuit interface, the signal quality can often be enhanced (and hence error rate reduced) by employing a "line driver" at each end of the cable; see the next section on local circuits for more information on these devices. Long runs of cable should employ shielded conductors or run the cable within conduit; this later option increases installation costs significantly but may be required by local fire protection codes anyway.

Local Circuits

Local "loop" circuits typically go off the immediate building premises and hence are usually leased from the local telephone company. They are categorized separately here because they are often limited in distance between their two endpoints and hence can utilize less expensive cable termination devices than those circuits involving longer distances; these devices are often referred to as "line drivers" or, equivalently, "limited distance modems". For this type of configuration we recommend ordering a type "LADS (metallic) 4 wire unloaded circuit" from your local telephone company. For a five mile circuit in San Francisco, tariff charges are $240 for installation and $70 per month lease. These circuits usually terminate in a connector block with four screw terminals for connecting the customer owned line driver units. Example synchronous line driver driver units include the Black Box LD421 (up to 19.2Kbps) at $370 and the Amdahl 983A DSU (56Kbps) at $900. Both of these units utilize synchronous mode data transfer and terminate on the host side in standard RS232 and CCITT V.35 circuit interfaces respectively. Note that transmission speeds are dependent on both the length and size (gauge) of wire used and this information should be obtained from the local telephone company and compared with the specifications for the particular equipment you have selected before either the LADS circuit or the line driver equipment is ordered. Some line drivers are capable of running at reduced rates so that a circuit that is too long to run at a high data rate may perform adquately at a lower rate.

Distance Circuits

Circuits longer than 5 miles must usually be leased from the telephone company and costs vary significantly depending on distance between endpoints and speed of data transmission. We present three relatively simple configurations:

1) For 9.6Kbps we suggest a type "3002" dedicated 4-wire circuit with possibly "C1" conditioning and with each end of the circuit terminating in a synchronous modem.

2) For 56Kbps we suggest a type "DDS" (Digital Data Service) circuit with each end terminated by a "DSU" (Data Service Unit).

3) For 1.5Mbps we suggest a type "T1" digital circuit with each end terminated by a digital "channel bank". Be warned, however, that there are few host interfaces available for this high data rate.

Costs for each of these configurations depends upon local telephone company tariff charges and the distance between endpoints. For comparative purposes we summarize the costs associated for the link between UCSF and UCB (13.3 miles line-of-site distance):

Speed	Circuit Costs		Equipment Costs	
	Installation	Monthly	Modems (2)	Interfaces (2)
9.6Kbps	$358	$144	$2,790[2]	$11,730[5]
56Kbps	$1,780	$1,338	$1,790[3]	$11,730[5]
1.5Mbps	$5,505	$2,150	$8,170[4]	$13,800[6]
T1 μwave	$36,000[1]	-0-	$8,170[4]	$13,800[6]

Table 1. Comparative Costs for Leased Circuit from UCSF to UCB
(Reference numbers shown next to prices refer to equipment listed in table 2)

Table 1 includes purchase costs for an example T1 terrestrial microwave link. In urban areas where line-of-site distances between gateway hosts are less than twenty miles, these links can be a cost effective alternative to a leased circuit. The available bandwidth is substantial, and a multiplexor or "channel bank" is often employed to break up the T1 channel into 24 56Kbps circuits. Such a microwave link is currently being installed between UCSF and UCB at a total cost of ~$50,000, or less than two year's worth of leasing costs for a telephone company supplied T1 circuit.

Host Interfaces

Several host communications interfaces are available for DEC VAX and PDP-11 computers, including the popular DMR-11 (Unibus) and DMV-11 (Qbus) devices as well as ACC's new T1 based communications processor. Unfortunately, finding compatible high speed host interfaces for non-DEC machines is more difficult. Host interfaces typically require a data link level protocol such as SDLC, HDLC, or Xerox's new point-to-point protocol; furthermore, it is desirable that this protocol be implemented with on-board firmware in order to minimize host loading. Since we have little experience with multiple vendor protocol compatibility, we suggest you evaluate the marketplace carefully and use considerable caution when considering the purchase of any unproven configurations.

Software

There are a number of problems with the original 4.2BSD software related to supporting a dedicated TCP/IP internetwork link. The changes necessary to build a working system are outlined below; note, however, that all of these enchancements are included in the new 4.3BSD release:† 1) If you plan to use a DEC DMR-11 or DMC-11‡ network interface on a VAX or PDP-11 host, be forewarned that the network device driver included with 4.2BSD does not work. A new driver developed by Bill Nesheim at Cornell University and Lou Salkind at New York University should be used instead. 2) The /etc/ifconfig program must be modified to allow the setting of the destination address on a point-to-point link; it is desirable to also install a trivial bug fix to the kernel (routine ifunit() in if.c) to allow multiple interface ioctl calls to work properly. 3) If you want the routing daemon to properly broadcast routing information to the remote network across the point-to-point link, several changes must be made to the routing code; alternatively, routes can be manually installed using /etc/route. 4) For testing purposes it is strongly suggested that Mike Muuss' "ping" program be obtained*.

Testing

It is imperative that a means be provided for testing each independent element of a overall point-to-point dedicated link. Trouble is bound to arise sooner or later, and the time spent developing a test procedure before trouble strikes will be well worth it. The telephone company usually provides a method for verifying the correct functioning of its data circuit from a centralized test center, and likewise most modem manufacturers provide some sort of "loopback" mode where transmitted data can be fed immediately back into the modem receiver instead of being transmitted out onto the data line. Many modems also offer a remote loop back feature for looping back transmitted data at the remote modem instead of the local modem. Diagnostics for the host network interfaces are likewise important, and finally a user level UNIX program such the aforementioned "ping" program which sends and receives messages via standard Internet Control Message Protocol (ICMP) will also prove invaluable. The ping program can be used to both verify end-to-end Internet communications through any intervening gateway machines (thus testing host packet routing) and also to test the local gateway, network interface, line driver or

† Special arrangements for obtaining the new DMR-11 interface driver and diffs for /etc/ifconfig and /etc/routed can also be made by contacting this author (ucsfcgl!tef).

‡ The DMC-11 interface is an obsolete device and has several known hardware problems; you should avoid this interface if at all possible.

* This program is also distributed with 4.3BSD or is available directly from mike@brl.ARPA.

modem, and dedicated circuit. To use *ping* in this latter mode, set the local end network interface to have the same source and destination addresses; at UCSF we use an entry from /etc/hosts named "dmc-test" and set the address as follows:

```
ifconfig dmc0 down; route -f delete ucb-dmc ucsf-dmc
ifconfig dmc0 dmc-test dmc-test; route add dmc-test dmc-test 0
```

and then loop back all transmitted data with either the "analog (local) loopback" or the "remote loopback" test switches on the modem. The command

```
ping dmc-test
```

with then send ICMP ECHO packets out onto the local network interface. These packets will then immediately be looped back to the originating host, received and retransmitted as ECHO-REPLY packets, and ultimately received by the originating *ping* program. Since *ping* keeps track of statistics such as lost packets and round trip propagation time, this test method provides a convenient user level procedure for testing network interfaces, modems and circuits.

The point of all this is that there are many places along the data pathway where problems can arise, and with multiple vendors involved "finger pointing" is not uncommon. If you can isolate the segment of the link which is failing you can then replace the faulty component or at least convince the appropriate vendor that the problem lies within his equipment.

Vendor References

The table 2 provides additional information for equipment previously referenced in this paper. The list is not meant to be exhaustive and you should definitely shop around before buying.

Conclusions

A method of interconnecting local area networks utilizing gateway hosts and a dedicated point-to-point network link has been presented. Several possible alternatives for circuit configurations have been detailed as well. The work is based on a operational network link between two University of California campuses.

Manufacturer	Product	Model	Vendor Contact	Phone Number
Vitalink	Ethernet bridge	TransLAN	Vitalink Comm. Corp.	(415)968-5465
Paradyne	9.6Kb modem	Challenger	Paradyne	(813)530-2000
Fujitsu[2]	9.6Kb modem	1921L	Interlink	(408)720-8838
Black Box	9.6Kb line driver	LD421	Black Box Corp.	(412)746-5530
Amdahl[3]	56Kb DSU	983A-631	Interlink	(408)720-8838
DEC[5]	Sync interface	DMR-11	Digital Equipment	(800)672-3414
ACC[6]	T1 interface	ACP6100	Advanced Computer	(805)963-9431
Farinon[1]	μwave radio	Urbanet 10	Harris/Farinon Corp.	(800)327-4666
Coast Comm[4]	T1 channel bank	90301-101	Coast Communications	(415)825-7500

Table 2. Vendor References
(Reference numbers refer to data in table 1).

Acknowledgements

I thank Mike Karels at UCB for pursuing the many complex issues and problems with the original 4.2BSD routing daemon code as it related to point-to-point network links. I also thank Kirk McKusick for supplying me with the inspiration to document my experiences in a more formal form and Laurie Jarvis for her assistance with PIC. This work was supported by the National Institutes of Health, Division of Research Resources, grant number RR-1081.

Design and Implementation of the Sun Network Filesystem

Russel Sandberg
David Goldberg
Steve Kleiman
Dan Walsh
Bob Lyon

Sun Microsystems, Inc.
2550 Garcia Ave.
Mountain View, CA. 94110
(415) 960–7293

Introduction

The Sun Network Filesystem (NFS) provides transparent, remote access to filesystems. Unlike many other remote filesystem implementations under UNIX†, the NFS is designed to be easily portable to other operating systems and machine architectures. It uses an External Data Representation (XDR) specification to describe protocols in a machine and system independent way. The NFS is implemented on top of a Remote Procedure Call package (RPC) to help simplify protocol definition, implementation, and maintenance.

In order to build the NFS into the UNIX 4.2 kernel in a user transparent way, we decided to add a new interface to the kernel which separates generic filesystem operations from specific filesystem implementations. The "filesystem interface" consists of two parts: the Virtual File System (VFS) interface defines the operations that can be done on a filesystem, while the vnode interface defines the operations that can be done on a file within that filesystem. This new interface allows us to implement and install new filesystems in much the same way as new device drivers are added to the kernel.

In this paper we discuss the design and implementation of the filesystem interface in the kernel and the NFS virtual filesystem. We describe some interesting design issues and how they were resolved, and point out some of the shortcomings of the current implementation. We conclude with some ideas for future enhancements.

Design Goals

The NFS was designed to make sharing of filesystem resources in a network of non-homogeneous machines easier. Our goal was to provide a UNIX-like way of making remote files available to local programs without having to modify, or even recompile, those programs. In addition, we wanted remote file access to be comparable in speed to local file access.

The overall design goals of the NFS were:

Machine and Operating System Independence
: The protocols used should be independent of UNIX so that an NFS server can supply files to many different types of clients. The protocols should also be simple enough that they can be implemented on low end machines like the PC.

Crash Recovery
: When clients can mount remote filesystems from many different servers it is very important that clients be able to recover easily from server crashes.

Transparent Access
: We want to provide a system which allows programs to access remote files in exactly the same way as local files. No pathname parsing, no special libraries, no recompiling. Programs should not be able to tell whether a file is remote or local.

† UNIX is a trademark of Bell Laboratories.

UNIX Semantics Maintained on Client
: In order for transparent access to work on UNIX machines, UNIX filesystem semantics have to be maintained for remote files.

Reasonable Performance
: People will not want to use the NFS if it is no faster than the existing networking utilities, such as *rcp*, even if it is easier to use. Our design goal is to make NFS as fast as the Sun Network Disk protocol (ND[1]), or about 80% as fast as a local disk.

Basic Design

The NFS design consists of three major pieces: the protocol, the server side and the client side.

NFS Protocol

The NFS protocol uses the Sun Remote Procedure Call (RPC) mechanism [1]. For the same reasons that procedure calls help simplify programs, RPC helps simplify the definition, organization, and implementation of remote services. The NFS protocol is defined in terms of a set of procedures, their arguments and results, and their effects. Remote procedure calls are synchronous, that is, the client blocks until the server has completed the call and returned the results. This makes RPC very easy to use since it behaves like a local procedure call.

The NFS uses a stateless protocol. The parameters to each procedure call contain all of the information necessary to complete the call, and the server does not keep track of any past requests. This makes crash recovery very easy; when a server crashes, the client resends NFS requests until a response is received, and the server does no crash recovery at all. When a client crashes no recovery is necessary for either the client or the server. When state is maintained on the server, on the other hand, recovery is much harder. Both client and server need to be able to reliably detect crashes. The server needs to detect client crashes so that it can discard any state it is holding for the client, and the client must detect server crashes so that it can rebuild the server's state.

Using a stateless protocol allows us to avoid complex crash recovery and simplifies the protocol. If a client just resends requests until a response is received, data will never be lost due to a server crash. In fact the client can not tell the difference between a server that has crashed and recovered, and a server that is slow.

Sun's remote procedure call package is designed to be transport independent. New transport protocols can be "plugged in" to the RPC implementation without affecting the higher level protocol code. The NFS uses the ARPA User Datagram Protocol (UDP) and Internet Protocol (IP) for its transport level. Since UDP is an unreliable datagram protocol, packets can get lost, but because the NFS protocol is stateless and the NFS requests are idempotent, the client can recover by retrying the call until the packet gets through.

The most common NFS procedure parameter is a structure called a file handle (fhandle or fh) which is provided by the server and used by the client to reference a file. The fhandle is opaque, that is, the client never looks at the contents of the fhandle, but uses it when operations are done on that file.

An outline of the NFS protocol procedures is given below. For the complete specification see the *Sun Network Filesystem Protocol Specification [2].*

null() returns ()
: Do nothing procedure to ping the server and measure round trip time.

lookup(dirfh, name) returns (fh, attr)
: Returns a new fhandle and attributes for the named file in a directory.

create(dirfh, name, attr) returns (newfh, attr)
: Creates a new file and returns its fhandle and attributes.

remove(dirfh, name) returns (status)
: Removes a file from a directory.

getattr(fh) returns (attr)
: Returns file attributes. This procedure is like a stat call.

[1] ND, the Sun Network Disk Protocol, provides block-level access to remote, sub-partitioned disks.

setattr(fh, attr) returns (attr)
Sets the mode, uid, gid, size, access time, and modify time of a file. Setting the size to zero truncates the file.

read(fh, offset, count) returns (attr, data)
Returns up to *count* bytes of data from a file starting *offset* bytes into the file. **read** also returns the attributes of the file.

write(fh, offset, count, data) returns (attr)
Writes *count* bytes of data to a file beginning *offset* bytes from the beginning of the file. Returns the attributes of the file after the write takes place.

rename(dirfh, name, tofh, toname) returns (status)
Renames the file *name* in the directory *dirfh*, to *toname* in the directory *tofh*.

link(dirfh, name, tofh, toname) returns (status)
Creates the file *toname* in the directory *tofh*, which is a link to the file *name* in the directory *dirfh*.

symlink(dirfh, name, string) returns (status)
Creates a symbolic link *name* in the directory *dirfh* with value *string*. The server does not interpret the *string* argument in any way, just saves it and makes an association to the new symbolic link file.

readlink(fh) returns (string)
Returns the string which is associated with the symbolic link file.

mkdir(dirfh, name, attr) returns (fh, newattr)
Creates a new directory *name* in the directory *dirfh* and returns the new fhandle and attributes.

rmdir(dirfh, name) returns(status)
Removes the empty directory *name* from the parent directory *dirfh*.

readdir(dirfh, cookie, count) returns(entries)
Returns up to *count* bytes of directory entries from the directory *dirfh*. Each entry contains a file name, file id, and an opaque pointer to the next directory entry called a *cookie*. The *cookie* is used in subsequent **readdir** calls to start reading at a specific entry in the directory. A **readdir** call with the *cookie* of zero returns entries starting with the first entry in the directory.

statfs(fh) returns (fsstats)
Returns filesystem information such as block size, number of free blocks, etc.

New fhandles are returned by the **lookup**, **create**, and **mkdir** procedures which also take an fhandle as an argument. The first remote fhandle, for the root of a filesystem, is obtained by the client using another RPC based protocol. The MOUNT protocol takes a directory pathname and returns an fhandle if the client has access permission to the filesystem which contains that directory. The reason for making this a separate protocol is that this makes it easier to plug in new filesystem access checking methods, and it separates out the operating system dependent aspects of the protocol. Note that the MOUNT protocol is the only place that UNIX pathnames are passed to the server. In other operating system implementations the MOUNT protocol can be replaced without having to change the NFS protocol.

The NFS protocol and RPC are built on top of an External Data Representation (XDR) specification [3]. XDR defines the size, bytes order and alignment of basic data types such as string, integer, union, boolean and array. Complex structures can be built from the basic data types. Using XDR not only makes protocols machine and language independent, it also makes them easy to define. The arguments and results of RPC procedures are defined using an XDR data definition language that looks a lot like C declarations.

Server Side

Because the NFS server is stateless, as mentioned above, when servicing an NFS request it must commit any modified data to stable storage before returning results. The implication for UNIX based servers is that requests which modify the filesystem must flush all modified data to disk before returning from the call. This means that, for example on a **write** request, not only the data block, but also any modified indirect blocks and the block containing the inode must be flushed if they have been modified.

Another modification to UNIX necessary to make the server work is the addition of a generation number in the inode, and a filesystem id in the superblock. These extra numbers make it possible for the server to use the inode number, inode generation number, and filesystem id

together as the fhandle for a file. The inode generation number is necessary because the server may hand out an fhandle with an inode number of a file that is later removed and the inode reused. When the original fhandle comes back, the server must be able to tell that this inode number now refers to a different file. The generation number has to be incremented every time the inode is freed.

Client Side

The client side provides the transparent interface to the NFS. To make transparent access to remote files work we had to use a method of locating remote files that does not change the structure of path names. Some UNIX based remote file access schemes use *host:path* to name remote files. This does not allow real transparent access since existing programs that parse pathnames have to be modified.

Rather than doing a "late binding" of file address, we decided to do the hostname lookup and file address binding once per filesystem by allowing the client to attach a remote filesystem to a directory using the *mount* program. This method has the advantage that the client only has to deal with hostnames once, at mount time. It also allows the server to limit access to filesystems by checking client credentials. The disadvantage is that remote files are not available to the client until a mount is done.

Transparent access to different types of filesystems mounted on a single machine is provided by a new filesystems interface in the kernel. Each "filesystem type" supports two sets of operations: the Virtual Filesystem (VFS) interface defines the procedures that operate on the filesystem as a whole; and the Virtual Node (vnode) interface defines the procedures that operate on an individual file within that filesystem type. Figure 1 is a schematic diagram of the filesystem interface and how the NFS uses it.

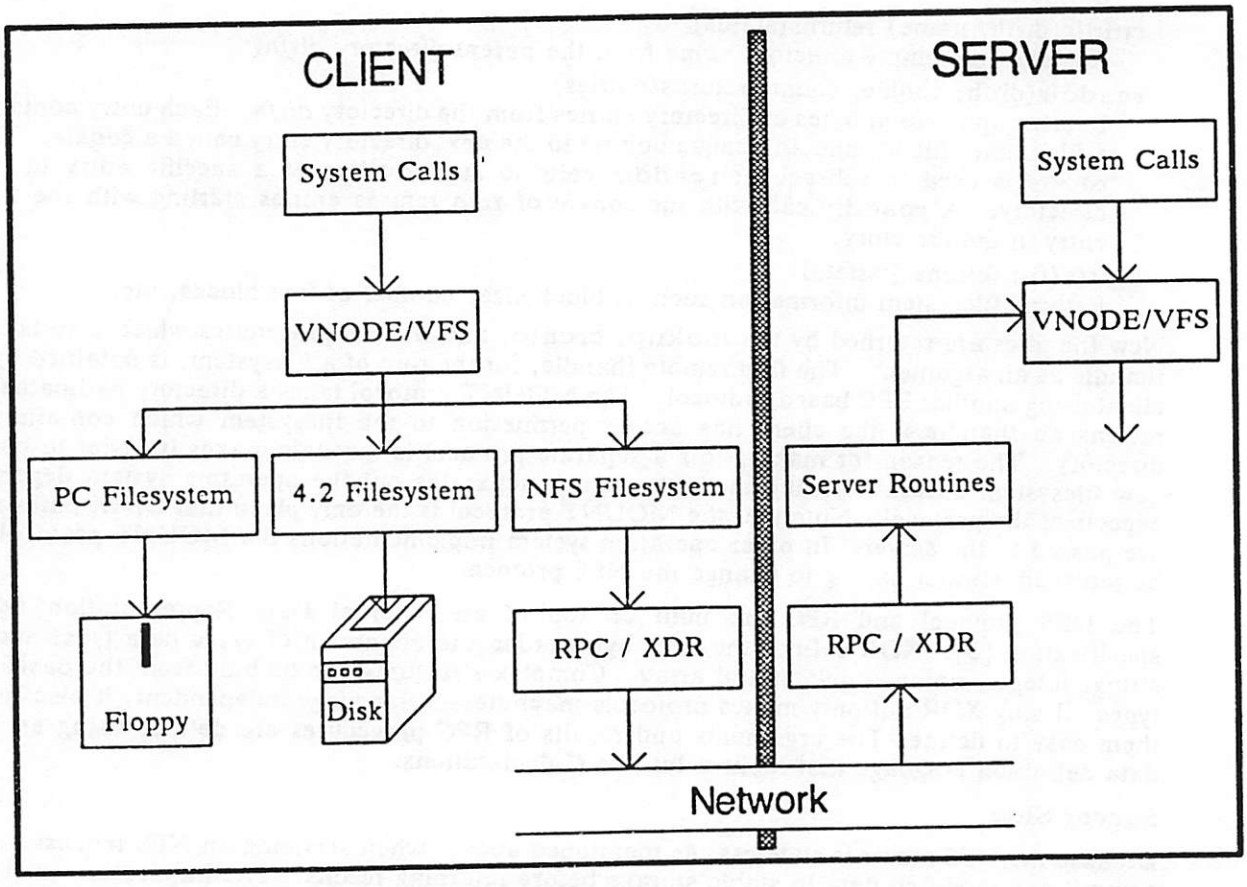


Figure 1

The Filesystem Interface

The VFS interface is implemented using a structure that contains the operations that can be done on a whole filesystem. Likewise, the vnode interface is a structure that contains the operations that can be done on a node (file or directory) within a filesystem. There is one VFS structure per

mounted filesystem in the kernel and one vnode structure for each active node. Using this abstract data type implementation allows the kernel to treat all filesystems and nodes in the same way without knowing which underlying filesystem implementation it is using.

Each vnode contains a pointer to its parent VFS and a pointer to a mounted-on VFS. This means that any node in a filesystem tree can be a mount point for another filesystem. A **root** operation is provided in the VFS to return the root vnode of a mounted filesystem. This is used by the pathname traversal routines in the kernel to bridge mount points. The **root** operation is used instead of just keeping a pointer so that the root vnode for each mounted filesystem can be released. The VFS of a mounted filesystem also contains a back pointer to the vnode on which it is mounted so that pathnames that include ".." can also be traversed across mount points.

In addition to the VFS and vnode operations, each filesystem type must provide **mount** and **mount_root** operations to mount normal and root filesystems. The operations defined for the filesystem interface are:

Filesystem Operations

mount(varies)	System call to mount filesystem
mount_root( )	Mount filesystem as root

VFS Operations

unmount(vfs)	Unmount filesystem
root(vfs) returns(vnode)	Return the vnode of the filesystem root
statfs(vfs) returns(fsstatbuf)	Return filesystem statistics
sync(vfs)	Flush delayed write blocks

Vnode Operations

open(vnode, flags)	Mark file open
close(vnode, flags)	Mark file closed
rdwr(vnode, uio, rwflag, flags)	Read or write a file
ioctl(vnode, cmd, data, rwflag)	Do I/O control operation
select(vnode, rwflag)	Do select
getattr(vnode) returns(attr)	Return file attributes
setattr(vnode, attr)	Set file attributes
access(vnode, mode)	Check access permission
lookup(dvnode, name) returns(vnode)	Look up file name in a directory
create(dvnode, name, attr, excl, mode) returns(vnode)	Create a file
remove(dvnode, name)	Remove a file name from a directory
link(vnode, todvnode, toname)	Link to a file
rename(dvnode, name, todvnode, toname)	Rename a file
mkdir(dvnode, name, attr) returns(dvnode)	Create a directory
rmdir(dvnode, name)	Remove a directory
readdir(dvnode) returns(entries)	Read directory entries
symlink(dvnode, name, attr, to_name)	Create a symbolic link
readlink(vp) returns(data)	Read the value of a symbolic link
fsync(vnode)	Flush dirty blocks of a file
inactive(vnode)	Mark vnode inactive and do clean up
bmap(vnode, blk) returns(devnode, mappedblk)	Map block number
strategy(bp)	Read and write filesystem blocks
bread(vnode, blockno) returns(buf)	Read a block
brelse(vnode, buf)	Release a block buffer

Notice that many of the vnode procedures map one-to-one with NFS protocol procedures, while other, UNIX dependent procedures such as **open**, **close**, and **ioctl** do not. The **bmap**, **strategy**, **bread**, and **brelse** procedures are used to do reading and writing using the buffer cache.

Pathname traversal is done in the kernel by breaking the path into directory components and doing a **lookup** call through the vnode for each component. At first glance it seems like a waste

of time to pass only one component with each call instead of passing the whole path and receiving back a target vnode. The main reason for this is that any component of the path could be a mount point for another filesystem, and the mount information is kept above the vnode implementation level. In the NFS filesystem, passing whole pathnames would force the server to keep track of all of the mount points of its clients in order to determine where to break the pathname and this would violate server statelessness. The inefficiency of looking up one component at a time is alleviated with a cache of directory vnodes.

Implementation

Implementation of the NFS started in March 1984. The first step in the implementation was modification of the 4.2 kernel to include the filesystem interface. By June we had the first "vnode kernel" running. We did some benchmarks to test the amount of overhead added by the extra interface. It turned out that in most cases the difference was not measurable, and in the worst case the kernel had only slowed down by about 2%. Most of the work in adding the new interface was in finding and fixing all of the places in the kernel that used inodes directly, and code that contained implicit knowledge of inodes or disk layout.

Only a few of the filesystem routines in the kernel had to be completely rewritten to use vnodes. *Namei*, the routine that does pathname lookup, was changed to use the vnode **lookup** operation, and cleaned up so that it doesn't use global state. The *direnter* routine, which adds new directory entries (used by **create**, **rename**, etc.), also had to be fixed because it depended on the global state from *namei*. *Direnter* also had to be modified to do directory locking during directory rename operations because inode locking is no longer available at this level, and vnodes are never locked.

To avoid having a fixed upper limit on the number of active vnode and VFS structures we added a memory allocator to the kernel so that these and other structures can be allocated and freed dynamically.

A new system call, *getdirentries*, was added to read directory entries from different types of filesystems. The 4.2 *readdir* library routine was modified to use the new system call so programs would not have to be rewritten. This change does, however, mean that programs that use *readdir* have to be relinked.

Beginning in March, the user level RPC and XDR libraries were ported to the kernel and we were able to make kernel to user and kernel to kernel RPC calls in June. We worked on RPC performance for about a month until the round trip time for a kernel to kernel **null** RPC call was 8.8 milliseconds. The performance tuning included several speed ups to the UDP and IP code in the kernel.

Once RPC and the vnode kernel were in place the implementation of NFS was simply a matter of writing the XDR routines to do the NFS protocol, implementing an RPC server for the NFS procedures in the kernel, and implementing a filesystem interface which translates vnode operations into NFS remote procedure calls. The first NFS kernel was up and running in mid August. At this point we had to make some modifications to the vnode interface to allow the NFS server to do synchronous **write** operations. This was necessary since unwritten blocks in the server's buffer cache are part of the "client's state".

Our first implementation of the MOUNT protocol was built into the NFS protocol. It wasn't until later that we broke the MOUNT protocol into a separate, user level RPC service. The MOUNT server is a user level daemon that is started automatically when a mount request comes in. It checks the file `/etc/exports` which contains a list of exported filesystems and the clients that can import them. If the client has import permission, the mount daemon does a **getfh** system call to convert a pathname into an fhandle which is returned to the client.

On the client side, the mount command was modified to take additional arguments including a filesystem type and options string. The filesystem type allows one *mount* command to mount any type of filesystem. The options string is used to pass optional flags to the different filesystem **mount** system calls. For example, the NFS allows two flavors of mount, soft and hard. A hard mounted filesystem will retry NFS calls forever if the server goes down, while a soft mount gives up after a while and returns an error. The problem with soft mounts is that most UNIX programs are not very good about checking return status from system calls so you can get some strange behavior when servers go down. A hard mounted filesystem, on the other hand, will never fail due to a server crash; it may cause processes to hang for a while, but data will not be lost.

In addition to the MOUNT server, we have added NFS server daemons. These are user level processes that make an **nfsd** system call into the kernel, and never return. This provides a user context to the kernel NFS server which allows the server to sleep. Similarly, the block I/O daemon, on the client side, is a user level process that lives in the kernel and services asynchronous block I/O requests. Because the RPC requests are blocking, a user context is necessary to wait for read-ahead and write-behind requests to complete. These daemons provide a temporary solution to the problem of handling parallel, synchronous requests in the kernel. In the future we hope to use a light-weight process mechanism in the kernel to handle these requests [4].

The NFS group started using the NFS in September, and spent the next six months working on performance enhancements and administrative tools to make the NFS easier to install and use. One of the advantages of the NFS was immediately obvious; as the *df* output below shows, a diskless workstation can have access to more than a Gigabyte of disk!

```
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/nd0                7445    5788     912    86%    /
/dev/ndp0               5691    2798    2323    55%    /pub
panic:/usr             27487   21398    3340    86%    /usr
fiat:/usr/src         345915  220122   91201    71%    /usr/src
panic:/usr/panic      148371  116505   17028    87%    /usr/panic
galaxy:/usr/galaxy      7429    5150    1536    77%    /usr/galaxy
mercury:/usr/mercury  301719  215179   56368    79%    /usr/mercury
opium:/usr/opium      327599   36392  258447    12%    /usr/opium
```

The Hard Issues

Several hard design issues were resolved during the development of the NFS. One of the toughest was deciding how we wanted to use the NFS. Lots of flexibility can lead to lots of confusion.

Root Filesystems

Our current NFS implementation does not allow shared NFS root filesystems. There are many hard problems associated with shared root filesystems that we just didn't have time to address. For example, many well-known, machine specific files are on the root filesystem, and too many programs use them. Also, sharing a root filesystem implies sharing `/tmp` and `/dev`. Sharing `/tmp` is a problem because programs create temporary files using their process id, which is not unique across machines. Sharing `/dev` requires a remote device access system. We considered allowing shared access to `/dev` by making operations on device nodes appear local. The problem with this simple solution is that many programs make special use of the ownership and permissions of device nodes.

Since every client has private storage (either real disk or ND) for the root filesystem, we were able to move machine specific files from shared filesystems into a new directory called `/private`, and replace those files with symbolic links. Things like `/usr/lib/crontab` and the whole directory `/usr/adm` have been moved. This allows clients to boot with only `/etc` and `/bin` executables local. The `/usr`, and other filesystems are then remote mounted.

Filesystem Naming

Servers export whole filesystems, but clients can mount any sub-directory of a remote filesystem on top of a local filesystem, or on top of another remote filesystem. In fact, a remote filesystem can be mounted more than once, and can even be mounted on another copy of itself! This means that clients can have different "names" for filesystems by mounting them in different places.

To alleviate some of the confusion we use a set of basic mounted filesystems on each machine and then let users add other filesystems on top of that. Remember though that this is just policy, there is no mechanism in the NFS to enforce this. User home directories are mounted on `/usr/servername`. This may seem like a violation of our goals because hostnames are now part of pathnames but in fact the directories could have been called `/usr/1`, `/usr/2`, etc. Using server names is just a convenience. This scheme makes workstations look more like timesharing terminals because a user can log in to any workstation and her home directory will be there. It also makes tilde expansion (~username is expanded to the user's home directory) in the C shell work in a network with many workstations.

To aviod the problems of loop detection and dynamic filesystem access checking, servers do not cross mount points on remote **lookup** requests. This means that in order to see the same

filesystem layout as a server, a client has to remote mount each of the server's exported filesystems.

Credentials, Authentication and Security

We wanted to use UNIX style permission checking on the server and client so that UNIX users would see very little difference between remote and local files. RPC allows different authentication parameters to be "plugged-in" to the packet header of each call so we were able to make the NFS use a UNIX flavor authenticator to pass uid, gid, and groups on each call. The server uses the authentication parameters to do permission checking as if the user making the call were doing the operation locally.

The problem with this authentication method is that the mapping from uid and gid to user must be the same on the server and client. This implies a flat uid, gid space over a whole local network. This is not acceptable in the long run and we are working on different authentication schemes. In the mean time, we have developed another RPC based service called the Yellow Pages (YP) to provide a simple, replicated database lookup service [5]. By letting YP handle `/etc/passwd` and `/etc/group` we make the flat uid space much easier to administrate.

Another issue related to client authentication is super-user access to remote files. It is not clear that the super-user on a workstation should have root access to files on a server machine through the NFS. To solve this problem the server maps user *root* (uid 0) to user *nobody* (uid –2) before checking access permission. This solves the problem but, unfortunately, causes some strange behavior for users logged in as *root*, since *root* may have fewer access rights to a file than a normal user.

Remote *root* access also affects programs which are set-uid *root* and need access to remote user files, for example *lpr*. To make these programs more likely to succeed we check on the client side for RPC calls that fail with EACCES and retry the call with the real-uid instead of the effective-uid. This is only done when the effective-uid is zero and the real-uid is something other than zero so normal users are not affected.

While restricting super-user access helps to protect remote files, the super-user on a client machine can still gain access by using *su* to change her effective-uid to the uid of the owner of a remote file.

Concurrent Access and File Locking

The NFS does not support remote file locking. We purposely did not include this as part of the protocol because we could not find a set of locking facilities that everyone agrees is correct. Instead we plan to build separate, RPC based file locking facilities. In this way people can use the locking facility with the flavor of their choice with minimal effort.

Related to the problem of file locking is concurrent access to remote files by multiple clients. In the local filesystem, file modifications are locked at the inode level. This prevents two processes writing to the same file from intermixing data on a single write. Since the server maintains no locks between requests, and a **write** may span several RPC requests, two clients writing to the same remote file may get intermixed data on long writes.

UNIX Open File Semantics

We tried very hard to make the NFS client obey UNIX filesystem semantics without modifying the server or the protocol. In some cases this was hard to do. For example, UNIX allows removal of open files. A process can open a file, then remove the directory entry for the file so that it has no name anywhere in the filesystem, and still read and write the file. This is a disgusting bit of UNIX trivia and at first we were just not going to support it, but it turns out that all of the programs that we didn't want to have to fix (*csh*, *sendmail*, etc.) use this for temporary files.

What we did to make open file removal work on remote files was check in the client VFS **remove** operation if the file is open, and if so **rename** it instead of removing it. This makes it (sort of) invisible to the client and still allows reading and writing. The client kernel then removes the new name when the vnode becomes inactive. We call this the 3/4 solution because if the client crashes between the **rename** and **remove** a garbage file is left on the server. An entry to *cron* can be added to clean up on the server.

Another problem associated with remote, open files is that access permission on the file can change while the file is open. In the local case the access permission is only checked when the file is opened, but in the remote case permission is checked on every NFS call. This means that if a client program opens a file, then changes the permission bits so that it no longer has read

permission, a subsequent **read** request will fail. To get around this problem we save the client credentials in the file table at open time, and use them in later file access requests.

Not all of the UNIX open file semantics have been preserved because interactions between two clients using the same remote file can not be controlled on a single client. For example, if one client opens a file and another client removes that file, the first client's **read** request will fail even though the file is still open.

Time Skew

Time skew between two clients or a client and a server can cause time associated with a file to be inconsistent. For example, *ranlib* saves the current time in a library entry, and *ld* checks the modify time of the library against the time saved in the library. When *ranlib* is run on a remote file the modify time comes from the server while the current time that gets saved in the library comes from the client. If the server's time is far ahead of the client's it looks to *ld* like the library is out of date. There were only three programs that we found that were affected by this, *ranlib*, *ls* and *emacs*, so we fixed them.

This is a potential problem for any program that compares system time to file modification time. We plan to fix this by limiting the time skew between machines with a time synchronization protocol.

Performance

The final hard issue is the one everyone is most interested in, performance.

Much of the time since the NFS first came up has been spent in improving performance. Our goal was to make NFS faster than the ND in the 1.1 Sun release (about 80% of the speed of a local disk). The speed we are interested in is not raw throughput, but how long it takes to do normal work. To track our improvements we used a set of benchmarks that include a small C compile, tbl, nroff, large compile, f77 compile, bubble sort, matrix inversion, make, and pipeline.

The graph below shows the speed of the first NFS kernel compared to various disks on the 1.1 release of the kernel. The NFS and ND benchmarks were run using a Sun-2 (68010 running at 10 Mhz with no wait states) model 100U for the client machine, and a Sun-2 120 for the server, with Sun 10 Megabit ethernet boards. The disk benchmarks were done on a Fujitsu Eagle with a Xylogics 450 controller, and a Micropolis 42-Megabyte drive with a SCSI controller. Notice that NFS performance is pretty bad, except in the case of matrix inversion, because there is essentially no filesystem work going on.

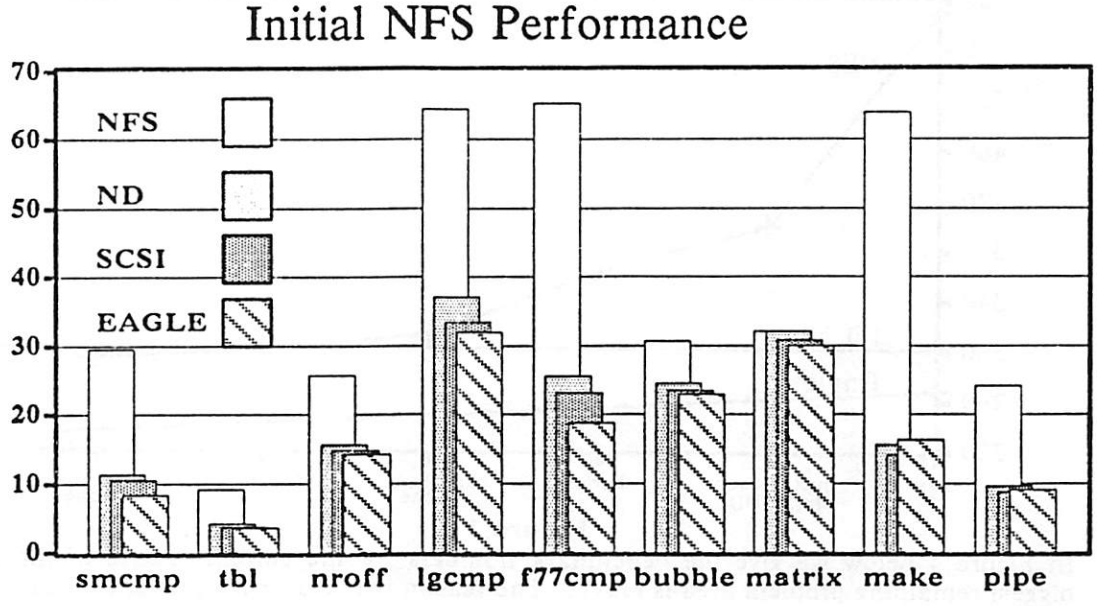


Figure 2

In our first attempt to increase performance we added buffer caching on the client side to decrease the number of **read** and **write** requests going to the server. To maintain cache

consistency, files are flushed on close. This helped a lot, but in reading and writing large files there were still too many requests going to the server. We were able to decrease the number of requests by changing the maximum UDP packet size from 2048 bytes to 9000 bytes (8k requests plus some overhead). This allows the NFS to send one large request and let the IP layer fragment and reassemble the packet. With a little work on the IP fragmentation code this turns out to be a big win in terms of raw throughput.

A *gprof* of the kernel, both client and server sides, showed that *bcopy* was a big consumer because the NFS and RPC kernel code caused three *bcopy*s on each side. We managed to trim that down to two copies on each side by doing XDR translation directly into, and out of, *mbuf* chains.

Using statistics gathered on the NFS server, we noticed that **getattr** (stat) accounted for 90% of the calls made to the server. In fact, the stat system call itself caused eleven RPC requests, seven of which were **getattr** requests on the same file. To speed up **getattr** we added a client side attribute cache. The cache is updated every time new attributes arrive from the server, and entries are discarded after three seconds for files or thirty seconds for directories. This caused the number of **getattr** requests to drop to about 10% of the total calls.

To make sequential read faster we added read-ahead in the server. This helped somewhat but it was noted that most of the **read** requests being done were in demand-loading executables, and these were not benefiting from read-ahead. To improve loading of executables we use two tricks. First, fill-on-demand clustering is used to group many small page-in requests into one large one. The second trick takes advantage of the fact that most small programs touch all of their pages before exiting. We treat 413 (paged in) programs as 410 (swapped in) if they are smaller than a fixed threshold size. This speeds up both the local and remote filesystems because loading a small program happens all at once, which allows read-ahead. This may sound like a hack but it can be thought of as a better initial estimate of the working set of small programs, since small programs are more likely to use all of their pages than none of them.

To make lookup faster we decided to add yet another cache to the client side. The directory name lookup cache holds the vnodes for remote directory names. This helps speed up programs that reference many files with the same initial path. The directory cache is flushed when the attributes returned in a NFS request do not match the attributes of the cached vnode.

Figure 3 shows the performance over the whole set of benchmarks for NFS compared to our performance goal (ND in the 1.1 release) and to an Eagle disk. Notice that the Eagle also got faster as a result of these improvements.

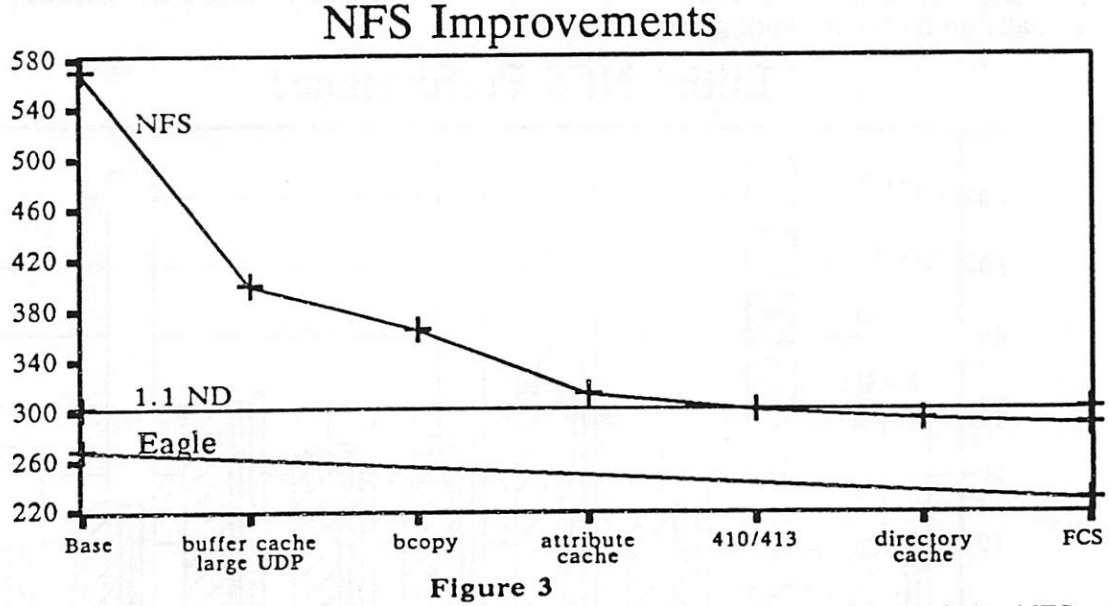


Figure 3

In Figure 4 below we give the benchmark numbers for the current release of the NFS. The biggest remaining problem area is *make*. The reason is that *stat*-ing lots of files causes one RPC call to the server for each file. In the local case the inodes for the whole directory end up in the buffer cache and then *stat* is just a memory reference. The other operation that is slow is **write** because it is synchronous on the server. Fortunately, the number of **write** calls in normal use is very small (about 5% of all calls to the server) so it is not noticeable unless the client does a large

write to a remote file. To speed up *make* we are considering modifying the **getattr** operation to return attributes for multiple files in one call.

Since many people in the UNIX community base performance estimates on raw transfer speed we also measured those. The current numbers on raw transfer speed are: 120 kilobytes/second for read (`cp bigfile /dev/null`) and 40 kilobytes/second for write. Figure 4, below, shows the same set of benchmarks as in Figure 2, this time run with the current NFS release.

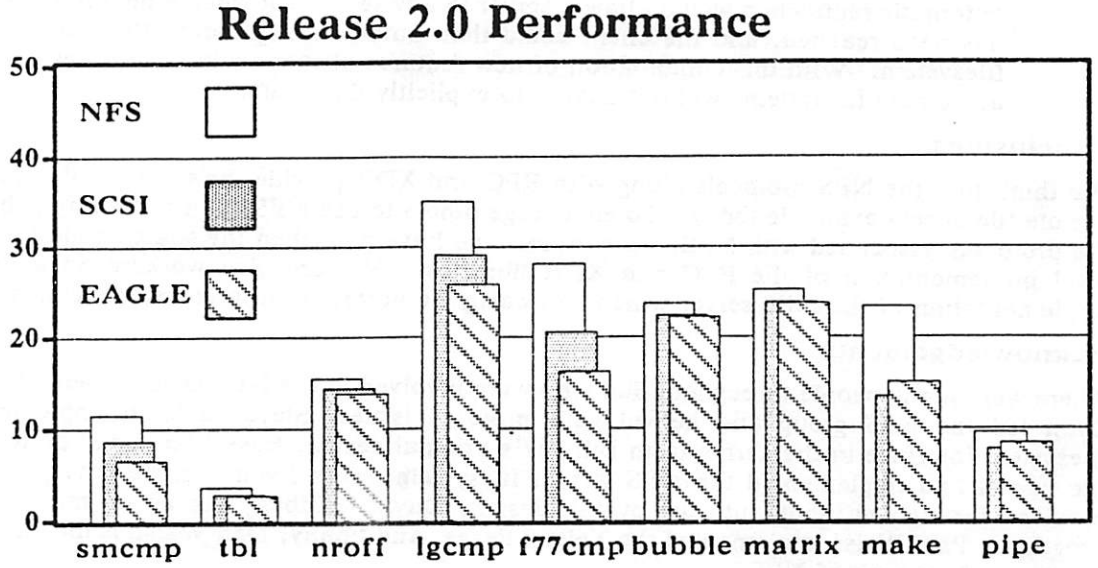


Figure 4

What Next?

These are some of the outstanding issues and new features of NFS that we will be working on in the future:

Full Diskless Operation
: One of the biggest problems with the current release is that diskless machines must use both ND and NFS. This makes administration twice as hard as it should be, and also makes our job twice as hard since we must support two protocols. We will be working on TFTP booting and additions to NFS to allow shared root filesystems, shared remote swapping, and remote device access. Together these will allow us to run diskless clients with only one remote file access method.

Remote File Locking
: We plan to build remote file locking services that are separate from the NFS service. Since file locking is inherently stateful (the server maintains the lock information) it will be built using the Sun status monitor service [6].

Other Filesystem Types
: The filesystem types that we have implemented so far are 4.2, NFS and a MS/DOS filesystem that runs on a floppy. We have barely scratched the surface of the usefulness of the filesystem interface. The interface could be used, for example, to implement filesystems to allow UNIX access to VMS or System V disk packs.

Performance
: We will continue our work on increasing performance, in particular, we plan to explore hardware enhancements to the server side since the server CPU speed is the bottleneck in the current implementation. We are currently considering building a low cost, stand-alone NFS server that would use a new filesystem type for higher performance and to allow automatic repair without operator intervention.

Better Security
: The NFS, like most network services, is prone to security problems because

programs can be written that impersonate a server. There are also problems in the current implementation of the NFS with clients impersonating other clients. To improve security, we plan to build a better authentication scheme that uses public key encryption.

Automatic Mounting

We are considering building a new filesystem type which would give access to all of the exported filesystems in the network. The root directory would contain a directory for each accessable, remote filesystem. Adding protocol support for automatic redirection would allow a server to advise a client when a mount point has been reached, and the client could then automatically mount that remote filesystem. With this combination of new features clients could have access to all remote filesystems without having to explicitly do mounts.

Conclusions

We think that the NFS protocols along with RPC and XDR provide the most flexible method of remote file access available today. To encourage others to use NFS, Sun is making public all of the protocols associated with NFS. In addition, we have published the source code for the user level implementation of the RPC and XDR libraries. We are also working on a user level implementation of the NFS server which can easily be ported to different architectures.

Acknowledgements

There were many people throughout Sun who were involved in the NFS development effort. Bob Lyon led the NFS group and helped with protocol issues, Steve Kleiman implemented the filesystem interface in the kernel from Bill Joy's original design, Russel Sandberg ported RPC to the kernel and implemented the NFS virtual filesystem, Tom Lyon designed the protocol and provided far sighted inputs into the overall design, David Goldberg worked on many user level programs, Paul Weiss implemented the Yellow Pages, and finally, Dan Walsh is the one to thank for the performance of NFS.

I would like to thank Interleaf for making it possible for me to write this paper without using troff!

References

[1] B. Lyon, "Sun Remote Procedure Call Specification," Sun Microsystems, Inc. Technical Report, (1984).

[2] R. Sandberg, "Sun Network Filesystem Protocol Specification," Sun Microsystems, Inc. Technical Report, (1985).

[3] B. Lyon, "Sun External Data Representation Specification," Sun Microsystems, Inc. Technical Report, (1984).

[4] J. Kepecs, "Lightweight Processes for UNIX Implementation and Applications," Usenix (1985)

[5] P. Weiss, "Yellow Pages Protocol Specification," Sun Microsystems, Inc. Technical Report, (1985).

[6] J. M. Chang, "SunNet," Usenix (1985)

An Implementation of an Extended File System for UNIX

Clement T. Cole
Perry B. Flinn
Alan B. Atlas

MASSCOMP†
Software Engineering

ABSTRACT

This paper describes the development of an extended file system for MASSCOMP's Real Time Unix (RTU) operating system. It discusses a mechanism by which users can perform file operations upon data physically residing in backing store on a remote computer. All operations are transparent to processes running on the local host. Migration of processes to the remote computer was not considered as a goal and no attempt was made at solving that problem. This system is client/server based and operates between two or more MASSCOMP computers on the same Ethernet. These machines run an enhanced version of the RTU kernel with an enhanced version of the network software. A new reliable datagram protocol (RDP) that supports multiple connections through a single endpoint has been developed to simplify the kernel's interface to the communication mechanism and to improve the throughput of remote file operations. All remote file operations are based on transactions. A global protection domain is used to simplify the concept of file ownership; that is, a single set of *user-ids* and *group-ids* is used on all machines.

1. Introduction

Given a network of computers running Real-Time UNIX (RTU), the MASSCOMP variant of the UNIXRit74a Time Sharing System, how can users of each machine on the network easily share databases? These databases might be anything from the system sources for a large programming project, to the telephone directory for an entire company. The key points are that the database is to be shared by many different programs and users throughout the network, and potentially could be *updated* by many different programs at different times. The programs used to update the shared database should be no different than those used to update a non-shared database. The goal of the MASSCOMP EFS project was to produce a new version of the operating system that would allow network-wide file operations without requiring modifications to existing user programs. Thus a user could easily and transparently share data between multiple machines.

† MASSCOMP and RTU are Trademarks of Massachusetts Computer Corporation.
UNIX is a Trademark of AT&T Bell Laboratories.

1.1 What is EFS?

EFS is the *Extended File System* facility for RTU that allows users to access data residing on remote backing store. The remote backing store is connected to a normal MASSCOMP machine operating with the EFS environment. Except for a small network time delay, any valid access to the data retained on the remote backing store is *indistinguishable* from an access to data retained within the backing store on the local host. This is referred to as *network transparency*. Note, however, that a new set of error codes was introduced to cope with the new types of errors that arise with the EFS environment.

1.2 Constraints

The UNIX system call interface standard proposed and accepted by the /usr/groupBuc84a and the draft standard of the IEEE P1003 POSE working groupIEEd)a have left a seemingly indelible mark on UNIX systems manufactured by different vendors. The standard interface dictates calling conventions and semantics for all major system calls, including file I/O operations. Therefore any attempt to produce an *extended file system* (EFS) for UNIX must lie within the boundaries set by this interface. Furthermore, the authors consider it non-optimal to force users to recompile or relink a working program when the operating system for the same computer hardware is changed from being a simple version of UNIX, to one that supports an EFS. The MASSCOMP EFS works within the constraints described above. All file operations (*open*(2), *close*(2), *read*(2), *write*(2), *ioctl*(2), etc.) continue to work as they did previously. Thus any program that works under a pre-EFS version of RTU continues to function correctly with an EFS version of RTU.

1.3 Why build an EFS?

In a timeshared system where all users perform their work on the same machine, data can easily be shared among users because it is all stored locally. Indeed, sharing is the norm. Most large systems go to great expense to allow users to share everything from files to in-core program images. In many cases, users may not even know they are working with objects shared by other users.

In the case of smaller workstations, however, data is stored on each of many machines. Since sharing is difficult, its occurrences become rare. In fact, data becomes replicated much more often than it is shared. Disks are copied either physically or via a network, but new versions of data from those disks are rarely sent to all sites that contain copies of it.

Consider the difficultly of keeping several workstations running the same version of system software, or worse yet, keeping many copies of some database up to date. An EFS allows workstations connected by a local area network to share data as though it were stored locally as it is in a large timeshared enironment.

For a more concrete example, consider an application such as the design of VLSI circuitry for a central processing unit. Graphical editors, such as KIC-2Bil83a or HAWKKel84a are used by IC designers to layout the circuit geometrically. The circuits are then simulated with tools such as SPICECoh76a, Qua83a and are finally converted to an IC mask. Along the development path, different engineers work on different pieces of the problem with different tools. Each engineer works on his or her piece of the circuit, yet each may periodically need access to other parts, or to the entire circuit as a whole. In order to help the entire design team work together, the master image of the circuit and the "cell libraries" are usually stored on one central machine. Each designer need only concern himself with the files associated with his portion of the project.

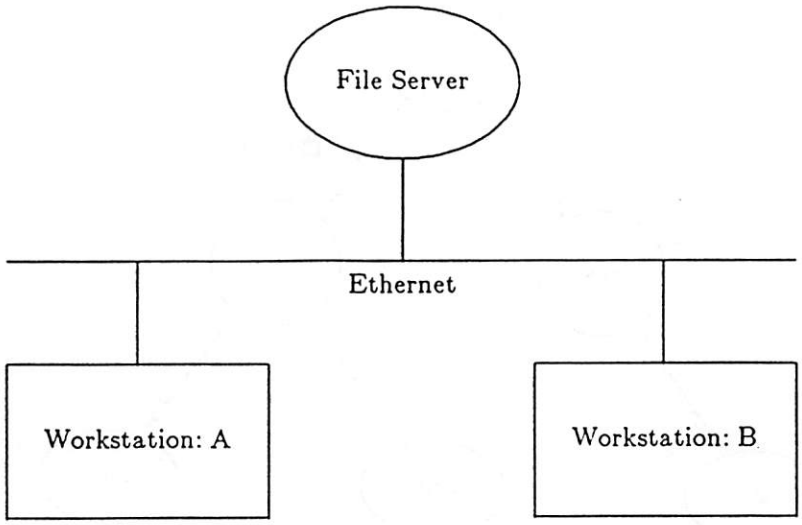


Figure: 1. Two design stations and a shared file server.

In figure 1, we see two different design stations sharing a common file server.[1] By letting each workstation remotely graft some part of the server's file system as part of its own directory hierarchy, any process running on a workstation, (such as the graphics editor) can access data on the remote file server *as though the data were stored locally.* The designer can then share common pieces, such as the cell libraries, without having to copy and store them locally. When the designer needs to use a cell, the editor many simply "read it in" from its master location on the file server.

Figure 2 shows two file systems "virtually linked" together so that each process on work station A views the remote directory hierarchy as part of its own. Cell libraries are contained on the remote file system, and the local system has a local copy of some part of the total circuit being designed.

1. Note, that you do not have to specify a "file server" as such. Any machine on the network that is participating in EFS can act as either a client or a server.

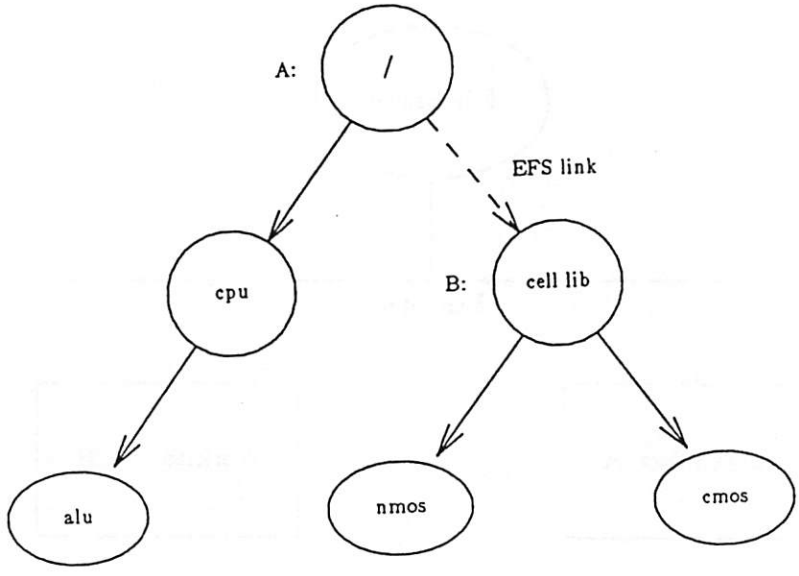


Figure: 2. Machine A is sharing part of Machine B's File System.

2. Previous Work

2.1 Version 8 File System

The goals of the MASSCOMP EFS project are similar to goals of the Version 8 File System described in the *Proceedings of the Summer 1984 USENIX conference*Wei84a and developed at Bell Laboratories. The Version 8 File System is proprietary to Bell Laboratories, and thus could not be used as a starting point for our efforts. The Version 8 File System, like EFS, is built around the *"inode"* concept.

2.2 Remote Virtual Disk

EFS differs from other network file system implementations in that the kernel has been modified to route the standard UNIX system calls to another host when necessary. It differs from the *remote virtual disk* (RVD) approach, as developed by Mike Greenwald and Larry AllenGre83a for the MIT Laboratory for Computer Science's multiple VAX/750 UNIX systems running as CPU servers. The MIT approach is similar to the LucasFilmDuf82a method. The major advantage of RVD is that under it, as under EFS, no user code must be modified. Unlike EFS however, RVD does not allow a given object-file or file system to be shared simultaneously by several machines. Rather it allows a single physical disk to be divided up into smaller chunks and parceled out to remote machines.

2.3 The Newcastle Connection

The work done by the University of Newcastle-upon-Tyne in EnglandBro82a was another approach that we chose not to follow. The Newcastle system is simple and has the advantage of being implemented with no kernel modifications; it is done instead as runtime library calls. The Newcastle system has the unfortunate side effect of starting up many child processes to do the network server work. Further, since the operating system does not "know" about the "remoteness" of a file, concepts like *change directory* are difficult to emulate.

2.4 NFS

Sun Microsystems' Network File System (NFS)Lyo85a is another implementation where the concept of remoteness is built into the kernel on a per-file basis. Whereas the main thrust behind NFS is portability to a variety of different machines and operating systems, EFS is aimed more at complete transparency and performance. This distinction in focus results in differences between the two designs. A prominent attribute of NFS is that it is designed to allow a server system to operate without having to retain intermediate state information between remote requests. This reduces implementation complexity for the server and simplifies error recovery. However, this comes at the expense of greater overhead per transaction and a consequent loss in performance, and a loss of transparency for certain types of remote operations.

3. The Design Space

The "design space" for our development effort imposed the following constraints.

1). All kernel code must be written to operate in a multiprocessor environment.

2.) EFS development must not impede any other operating system development.

3.) The design must operate using a "Client - Server" model.

4.) All operations revolve around *inodes*.

5.) Calls must be provided to enable and disable remote access.

6.) No single client process should be able block an EFS server from serving other client processes.

The next paragraphs describe these constraints and their impact on the design.

3.1 Multiprocessor Safety

Unlike other UNIX variants, notably AT&T's System V and the University of California's 4.2 BSD, RTU runs on both dual-processor and full multi-processor (MP) computer architectures. This constraint means that any new code added to the RTU kernel must not hinder the multiprocessor nature of the system.

3.2 Impact on other Development

At the time of the EFS development, three large kernel projects were underway, each of which entailed rewriting major sections of the MASSCOMP RTU kernel. The group working on EFS has become adept at folding changes into other versions of the kernel from other development groups. Overall, we have been reasonably successful, but that story is the subject for a different forum.

3.3 Clients and Servers

The operating model chosen is based on client-server style interactions. When a user process executes a system call locally, the file referenced by the call may be remote. In such cases, the local machine executes a "client" version of the operation that sends a message to the remote "server" machine. The server decodes the request, fulfills it if possible, and then returns the result to the client. The client process is blocked until the server completes the transaction.

3.4 *Inodes* and *Rinodes*

Inside the UNIX kernel, the focal point for all file operations is the *inode*.[2] This is a per-file structure that contains all of the information that UNIX keeps about the file, including its size, type, owner, protection, access and modification times, and location on the disk. While a file is being operated on, UNIX keeps a copy of its *inode* in-core. In general, the address of this in-core *inode* is the key used by UNIX system calls to gain access to the file.

In the EFS environment, an in-core *inode* address is insufficient to identify a remote file, since the regular *inode* can only describe files stored on the local system. We thus introduced the concept of a *remote inode*, or *rinode*, to describe remote files. An *rinode* is a special variant of an *inode*. Rather than containing the information that would be found in a normal *inode*, it contains a *machine-id*, which uniquely identifies a single host on the network, and the address of an in-core *inode* within the memory of that host. Where a remote file is involved, the *rinode* takes the place of the *inode* for system call operations.

3.5 Start Up and Take Down

To establish the mapping of a server's directory to the clients, a new system call was introduced: *rmount*(2). This call is analogous to the standard UNIX call, *mount*(2). Similarly, a new call was introduced to take down the connection namely: *rumount*(2). Again, there is an analogy with a standard UNIX call, *umount*(2).

3.6 Non-Blocking Server

Later in this paper, we describe the problems that can occur when a server process blocks. Simply, any client process must not cause the server to block in such a way that other client processes can not make requests to the server, or processes local to the server can not run.

4. The Implementation

The client machine can be thought of as the local host on which a user program executes. The server is the machine that is accessed by the local host to perform certain file I/O operations. All remote file operations are transactions from the client to the server. When the server receives an I/O request it takes all the information in the transaction and then operates upon that transaction. In certains cases, such as *write*(2), not all of the information is available to the server when the I/O request is sent. In those cases, the server sends a transaction back to the host requesting the needed information. All transactions return sucess or failure, and the user program is notified accordingly.

---

2. For a more complete discussion of the UNIX I/O system, seeRit78a

4.1 The *inode* and *namei*

In any operating system, there exists a method of mapping from a user's concept of the file specification to a pointer into the actual file system tables. Under the UNIX operating system, the user visible file specification is called a *pathname*, and the pointer into the file system is an *inode*. In the UNIX kernel, the routine called *namei* is used to convert from a *pathname* to an *inode*. It returns either a pointer to an *inode* or an error code. System calls that require a *pathname* as an argument such as *open*(2), *stat*(2), and *link*(2), internally call *namei* to perform a translation from the *pathname* to an *inode*.

To obtain a *transparent* file system, EFS operates on a new type of object, an *rinode*, which is a pointer into the file system on a remote machine. Thus *namei* must be able to return both *inodes* and *rinodes*. We have modified *namei* so that it walks a *pathname*, it checks each *inode* that it encounters for remoteness. This is indicated by a new bit in the *i_flag* field: IREMOTE.

To obtain this type of functionality, the path walk portion of *namei* includes new code, similar to:

```
if (inodePtr->i_flag & IREMOTE) {
    rinode = rnamei(ptrToRestOfPath);
    localInode = storeInInode(rinode);
    return(localInode);
} else {
    ... original namei() code ...
}
```

This new routine *rnamei* must look into the *rinode* and find the network handle for the remote system and put together all of the information for the remote procedure call.

In the pre-EFS system, when a user process wants to send data, it performs a *write*(2) system call which is mapped into the Ethernet driver. In the EFS based kernel, the *rnamei* calls the driver directly on a known[3] socket and performs a "remote procedure call" to the server process acting as the client's agent on the remote host.

Please notice that the read/write pointer is maintained by the client system, and not by the server, which implies that the open file pointer must be passed to the server each time an I/O operation is performed. Further, this implementation is an *inode* implementation not a *file* implementation, and no modifications to the normal UNIX open file table were required.

4.2 The Server Implementation

For the file system type system calls discussed in the client portions (*read*(2), *write*(2), *stat*(2), *link*(2), and etc.), the server must call its local system routine to obtain the needed information for the client. As an example, when the client calls its local *namei* routine and discovers it must call *rnamei*, the server picks up the *rnamei* request and calls its own *namei* routine to obtain a *remote specific inode*. This information is then sent back to the client so that client's *rnamei* routine can return the needed information to the client process that started this procedure in the first place.

For instance, when an *open*(2) takes place, the client does a *namei*, and then sends back to the server a message saying in effect: "open this file." The server has an incremented reference count for the *inode*. When the file is closed, the count is decremented.[4] Once referenced by the client, the file can be accessed by its *rinode* directly.

---

3. When the EFS is first started up the local host determines how to communicate with the remote. After the system is started, the communication information is stored locally so it can be used later for *namei/read/write* requests.

4. Incrementing the reference count does lead to error recovery problems which will be discussed later.

4.3 The Process Pool

The server contains a pool of available kernel processes that can act as agents for any EFS client request. This set of processes is known as the "process pool." This pool is a global resource and all EFS connections use the process pool to have their work performed. Furthermore, any *agent* process from the pool can act in the behalf of any client.

At EFS start up time, a master process on the server creates a socket at a well-known port[5] allowing client requests to be recieved. EFS This process is called the *lifeguard*. In general, the *lifeguard* is dormant. Its job is to dispatch incoming requests to idle agent processes, and manage error recovery.

Once a socket has been set up the system is ready to receive and handle client requests. When one is received, the *lifeguard* assigns the request to an idle *agent* process or puts it on queue until an agent is available to service the request. When a request comes in off the network:

1.) The lifeguard finds the first available agent;

2.) awakens the agent and passes the request to it.

3.) The agent copies information from the request to its *u*__area;

4.) invokes the "server" version of the requested system call;

5.) puts itself back into the available process pool as an inactive process;

6.) awaits a new request.

Notice that any process in the process pool can act as an agent for any request from any client.

In some cases the client will request more data than can be transmitted at once by the local area network. For this reason, a *more to come* flag was implemented in the returned packet. This notifies the client that more data should be expected. This feature is implemented at the lowest level of the request/reply packet handling routines.

4.3.1 Start Up and Take Down

Starting up EFS activity and taking it down is not quite as simple as with the standard UNIX *mount*(2) system call. As previously mentioned, a new call was introduced, *rmount*(2). This call is used to map a directory on the client machine into a directory on the server machine. Note that this is different from the *mount*(2) call which takes a "device name" on the local host and maps that into a directory on the local host.

When *rmount*(2) is called, the client contacts the master server process and requests that agents be made available for it. If all of the protection checks are passed, the master server sets up an active connection and allows transfers to continue. An agent process is given the request and return to the client a *network handle* with which all further communication takes place. From that point on, all EFS requests use that same *network handle* along with any file-specific data when requesting data of the server. Note that this handle must be stored with each *rinode* because it is possible for a local machine to be a client of more than one server, each on a different machine. That is to say, *the local machine may remotely mount many different directory hierarchies, and each remote file system request is performed by an agent processes running on its behalf, picked from the process pool on each server at each request.*

5. For details of communications see the sections on it. The concepts that are discussed are explained in detail in other literature.

It follows that the inverse operation must also be performed. A new call, *rumount*(2), was introduced to break out of a connection. As in *umount*(2), checks must be made to test if there are any active *inodes* from that client into that file tree. Since this call is not made often, a simple search is used to detect any server *inodes* that meet this criterion. If everything is clean, then the *rumount*(2) succeeds, otherwise an error is returned. As discussed later, there is a case where a *rumount*(2) is forced by error conditions. In that case, any file found active is made inactive.

5. Communications Support

Throughout the design of EFS, several assumptions were made about the attributes and capabilities of the underlying communications protocol. MASSCOMP supports an ethernet local-area-network using the DoD Internet protocol familyPos82a The programming interface to these protocols is the 4.2 BSD socket mechanismLef82a, Lef82b A primary assumption was that all communications between machines had to be carried out within the context of this mechanism.

Interactions between client and server systems are transaction-based. Most transactions consist of a single message requesting information or the performance of an operation, followed by a single reply supplying the requested information or the completion status of the operation. Others, most notably *read*(2) and *write*(2), may involve the transfer of large amounts of data in a single transaction. Thus, the communications protocol must provide a mechanism for grouping related messages. Further, since some operations may take longer than others to complete, either because of scheduling anomalies on the server or because the agent process must wait for some event, several transactions may be pending concurrently between a given pair of hosts.

Because transactions are so dynamic, the overhead of creating and destroying them must be low. To appreciate the importance of this requirement, consider that simply opening a remote file involves three separate transactions between client and server. This dynamism effectively rules out the possibility of using a protocol such as TCPPos81a which would require a separate socket and connection for each transaction.

In the process of carrying out remote operations, there are various intermediate states on both the client and the server in which network failures can have serious ill effects. Assume for example that an often used directory such as */tmp* becomes locked while a remote client is deleting a file from it. If in the midst of the operation the client system crashes, the server could slowly cease to function as processes queue up waiting to create or delete temporary files. It is clearly of vital importance that the communications mechanism provide reliable transfer of messages, and timely indication of communication failures.

To address these needs, a new *Reliable Datagram Protocol* (RDP) was designed.

5.1 RDP Functional Description

RDP provides a datagram-based communication service that guarantees in-order, reliable delivery of messages. That is, a successful return from a send operation implies that the message has been delivered to the peer RDP module on the destination host. An optional side effect of a send operation is the creation of a "connection" between the source and destination processes through which further related messages may be sent or received. The connection may be "duplex", meaning that both source and destination processes may transmit on the connection, or "simplex", meaning that only one or the other may transmit. Such connections are independent of any others that may also be in progress within the context of the same socket, and in fact, an arbitrary number may exist concurrently. A single process may manage several connections, or each of several processes sharing a common socket may manage a single connection.

When doing a receive operation, a process indicates whether it wants a message related to an existing connection, or one that is either not associated with a connection or is the first in a new one. Upon return from the receive, an indication is given about whether further messages may be received on the same connection, and whether or not reply messages may be sent.

Once a connection is established, each direction of transfer is under control of the sender. That is, the sender provides an *end-of-data* indication with the last message it sends, closing down one side of the connection. It may continue to receive from the other side, however, until the remote sender transmits its final message.

RDP provides a timeout mechanism to automatically abort a connection when the peer system crashes or some other communication failure arises. Idle connections are maintained by periodic "keep-alive" messages that are invisible to the communicating processes. Reliable delivery is ensured by positive acknowledgement of each message as it is received, and by periodic retransmission of unacknowledged messages.

5.2 RDP Implementation

RDP is implemented on top of the Internet Protocol (IP)Pos81b This choice was made primarily because it simplified the implementation by allowing us to take advantage of the considerable existing support framework for handling "Internet" addresses.

Access to RDP is provided through the 4.2 BSD socket mechanism using the "Internet" addressing domain and the SOCK_RDM socket type. RDP sockets are given local address bindings in the same manner as a UDP socket. The binding consists of a 16 bit port number and a 32 bit IP address.

Once the socket has been created and bound, a new address structure is used for send and receive operations. It is an extension of the standard "Internet" address structure:

```
struct sockaddr_rdp {
    short          sin_family;
    u_short        sin_port;
    struct in_addr sin_addr;
    u_long         sin_txid;
    u_long         sin_flags;
};
```

The first three fields correspond exactly to their *sockaddr_in* counterparts. The two additional fields, which overlay previously unused parts of a *sockaddr_in*, carry the information needed to manage RDP connections. The *sin_txid* field holds a 32 bit connection identifier that is unique among all other connections related to the same socket on the local machine. The *sin_flags* field holds two OR-able flags, SF_MORE and SF_REPLY, whose functions are described below.

5.2.1 Sending to an RDP Socket

When a send operation is performed on an RDP socket, the *sin_family*, *sin_port* and *sin_addr* fields of the destination address must be set up as they would be for a UDP socket. If the *sin_txid* field is non-zero, it must identify an existing writable connection.

A new connection is created when *sin_txid* is zero; upon return from the send, the zero is replaced by the new connection id. The bits in *sin_flags* determine whether a new connection will be simplex or duplex: SF_MORE alone creates a simplex connection from sender to receiver (i.e., in the direction of the original message); SF_REPLY alone creates a simplex connection in the opposite direction; SF_MORE and SF_REPLY together create a duplex connection. If neither bit is set, no connection is created, *sin_txid* remains zero, and the message is sent as a "standalone" datagram.

After a connection is created, the SF_MORE bit indicates that further messages are to be sent. Once a message is sent with this bit cleared, further attempts to send on the connection are rejected with an EPIPE error code (and possibly a SIGPIPE signal).

5.2.2 Receiving from an RDP Socket

In contrast to other protocols, the address structure passed to a receive operation on an RDP socket must be initialized before hand. In particular, the contents of the *sin_txid* field determine what effect the receive will have. If the field is non-zero, it must contain the identifier for an existing readable connection. Furthermore, the *sin_addr* and the *sin_port* fields must specify the remote address binding of the peer socket at the other end of the connection. In this case, the receive is satisfied only by a message sent on the specified connection; other pending messages directed to the same socket but on different connections remain queued.

If *sin_txid* contains zero, then only a message which is either the first in a new connection or is "standalone" (i.e., not associated with any connection) may be received. When the first message in a new connection is received, the returned *sin_txid* field contains the unique identifier for the connection. When a standalone message is received, the *sin_txid* field remains zero.

When any message is received on a connection (including the first message), *sin_flags* indicates connection status as follows:

SF_MORE indicates that further messages may be received on the connection. Once a message is received with this bit cleared, further attempts to receive on the connection return with a zero byte count.

SF_REPLY indicates whether messages may be sent on the connection. For a duplex connection, it is typically set unless the receiving process has previously sent a message on the connection without SF_MORE.

5.3 Development using UDP

Because the design and implementation of RDP proceeded in parallel with EFS implementation, much of the initial testing and debugging of EFS was performed using the User Datagram Protocol (UDP)Pos80a as a communication base. This was made possible by a conscious effort to isolate as much as possible the details of the network interface from the new EFS kernel code. This proved to be a successful plan. Very early on in the project, basic remote operations were being tested with UDP. In fact, even as the more complex operations were completed, UDP continued to serve as an adequate testing vehicle. Where UDP failed to satisfy EFS's communication needs was when the time came to test multiple concurrent operations. The difficulty here is that when a single socket is used for all EFS-related network traffic, UDP provides no mechanism for associating a reply message with its corresponding request. Thus, when two client processes on the same machine are waiting for replies from their EFS agent processes, nothing guarantees that the replies will be received by the right client.

6. Protection Issues

One of the trickier issues that must be discussed is protection across the EFS. In UNIX, a *user-name* (*clemc* for example) is mapped to a machine specific number, called a *user id* or *uid*. It is this number, not the name, that UNIX uses to determine a user's access rights. On one particular machine, XORN for example, the name *clemc* may be mapped to *uid* 92, while on another machine, say VAMPIRE, the *uid* may 25 instead.[6] By convention, user programs read the password file to obtain the *uid* from a *user-name* or vice-versa. Some network applications, such as the remote copy program, *rcp*(1), pass the *user-name* to the remote side, which in turn converts it to the corresponding local *uid*.

6. We have completely ignored the issue of users who have a different *user-name* on each machine. For instance, John Smith uses *johns* as his *user name* on the machine named *EARTH*, and uses *smith* as his *user name* on the machine named *MOON*.

In the case of EFS, when the two machines are able to share a file system, it would seem that they need to *share* a actual file *in some way*. Either they must share the password file, or the EFS software must have the server process convert each request from the *uid* from one client to a *uid* of the server.

Therefore, we wrote a program, *idupdate*(8), that works like *fsck*(8), and walks the file system in a standalone mode, and changes the *uid* from one to another. This serves as a conversion tool for customers who have the problem of different machines that do not agree with each other on the *uid*'s for certain users, (i.e. *clemc* not being 92 on all of the machines, for instance.)

After all of the machines in the network are converted to use the same password file, the normal UNIX protection mechanism may be used. This, however, is not enough. Without a change, this implementation would allow *Root* privileges on one machine to be maintained on the server. In most environments, there exist some files on the certain machines that should not be made accessible to everyone - including users from a remote machine with *root* privileges. For this reason, we added *selective mount points*. At *rmount*(8) time, the system administrator of the server has a file of mount points that a remote system is allowed to mount upon. In our VLSI example, that might be a path such as: /usr/vlsi/library. Any attempt by the client to mount elsewhere would fail.

Furthermore, because of the back pointer to the client used in pathname walking, (see the dot-dot problem described later) a program would not be able to subvert this mount point by performing a *change directory* to the remote and then using dot-dot style pathnames to get at data it should not have.

7. Error Recovery

With two hosts and two network interfaces involved, the number of failure modes is much greater than the usual single machine case. In addition, side effects of certain EFS transactions can cause problems ranging from the invisible and harmless to complete paralysis of the server. Error recovery is possible and has been included as part of the EFS.

7.1 Failure Modes

All EFS transactions are between two machines at a time, and error recovery can be analyzed for the two machine case even when fifty or more machines are all running EFS on the same network. In all cases, errors are treated as specific to the client-server pair of the current transaction.

The possible failure modes are:

1.) Client failure

2.) Client net failure

3.) Server failure

4.) Server net failure.

5.) Any combination thereof.

Client or server failure means an operating system failure such as a crash or power down. Net failure is the failure of the network interface or communications link without the host's failing also. This may result from intermittent cable faults or where the network is implemented in a coprocessor with a address space seperate from the host, a failure in that processor.

7.2 Side Effects

The client-server model used for EFS is not stateless. Once an *rmount*(2) has been done, information is required on both server and client to maintain the EFS relationship. Since EFS intervenes at the *inode* level in the kernel, some bookkeeping must be done to maintain reference counts sensibly. A *rinode* in-core on a client requires the associated *inode* on the server to be in-core there. This is accomplished by maintaining *inode* reference counts on the server for each client transaction on that *inode*. The effect is that if a client is exclusively using an *inode* on a server, the reference count on the server will mirror that on the client at all times. It is tricky but possible to achieve this, unless a failure occurs. Depending on which failure mode has occurred, a server may find itself with *inodes* in-core that are not being referenced by any local process but which have non-zero counts or a client may find itself with *rinodes* in-core which cannot be used since the server cannot be accessed.

A much more serious problem is the locked/unlocked problem. Inodes, being a shared resource, have a locking and unlocking mechanism that is used to prevent simultaneous access by two processes. Any process trying to access a locked *inode* blocks until it is unlocked. Certain EFS transactions leave *inodes* locked on the server, waiting for the next transaction from the client to complete the system call and free the *inode*. If a client failure occurs between these transactions, the server has locked *inodes* in-core which have been locked by a now-dead client.

7.3 Error Detection

There are two methods of error detection. Any client or server which receives an error return from a send or receive operation will activate a failure test immediately. If a failure of the local network interface is detected or if the operation times out (signifying the death of the other machine or its net interface), appropriate recovery procedures are instituted (see below). If not, the immediate system call or transaction still fails and returns an error code. It is simply too difficult (or impossible) to retry transactions at this point because of uncertainty as to exactly what happened before the failure.

The method of error detection just described is sufficient for the client since there are no side effects there that can cause damage in the absence of EFS activity. The case on the server is much different. Consider what happens when ten clients start ten transactions which leave ten *inodes* locked on the server, then each promptly fails or the network fails. The server must detect, in the absence of EFS traffic, that something is wrong and that certain *inodes* are locked on behalf of clients which could be unreachable right now. In order to detect this situation, the time of the last transaction from a given client is kept by the EFS *lifeguard* process. Periodically, a function is called which checks these "last transaction times" and sends a packet to any clients which haven't been heard from for a certain length of time. If no answer is received, the server recovers from that client (see below) and continues to check the other clients with which it has established a remote mount relationship until it has verified that all clients are alive.

7.4 Client Recovery

The simpler of the two cases by far, recovery on the client merely consists of identifying the offending server and unmounting all directories associated with that server. If the failure of the client's net interface is detected, then all *rmounts* are taken down. *Rinodes* which are being accessed by other processes on the client will each cause some net failure when accessed and will be cleared at that time.

7.5 Server Recovery

The ability of the server to recover is based on keeping information about *inode*s which are in-core on behalf of clients and which of those are locked. When a server process detects an error in trying to reach a particular client, it simply looks up all of the *inode*s which are in-core on behalf of that client and clears them out. If it is known *a priori* which *inode*s are locked, it is possible to circumvent the blocking problem mentioned above. Also, any *rmounts* which the dead client has are also removed from the server's *rmount* table.

Finally, each time a client tries to *rmount* to a particular server for (what the client considers) the first time (i.e., there are no *rmounts* on the client to that particular server) this belief is communicated to the server. If the server finds that it has *rmounts* from that client, it clears them out (and any *inode*s in-core from that client) before completing the current *rmount* request. This scheme corrects for the case where a client has failed and been rebooted before the server notices it.

8. Some Interesting Problems

In the course of development of EFS a number of problems were encountered that are generic to a file system that maps across multiple machines.

1.) The *core* file problem.

2.) The problem of server processes blocking on a client request.

3.) The problem of *pathnames* that cross "mount points" in a directory hierarchy.

4.) The *stat*(2) problem.

5.) The *ustat*(2) problem.

The next section will discuss each of these and how our implementation addressed each one.

8.1 The Core File Problem

One of the more interesting things to try on a machine that runs an EFS is to determine where the system should place the *core* file when an error arises that causes UNIX to create an image of the running process. This is a problem because of the way UNIX handles the *core* file. When UNIX decides to create a *core* file, it opens the current directory and creates the file called *core* in it. As it turns out, in our implementation, the *core*-file problem falls away.

Because the current directory is an *rinode* instead of a normal *inode*, the system does not see the difference and the normal UNIX mechanism takes place: UNIX opens a file in the current directory called *core*. The open is done with the *inode* of the current directory which in this case happens to be an *rinode*, the same basic frame work applies.

8.2 The Blocking Problem

As mentioned earlier, an EFS server maintains a pool of processes to act as agents for remote clients. The astute reader might ask, "Why not use a single agent process for each active EFS connection?" Arnovitz articulated the question further by asking: "If a single process is used as an agent, what happens if that process needs to block?"Arn84a The answer for a single agent process is that if the agent process must block, then all other requests from the client host will have to block waiting until the first request completes.

By using a pool of processes, an agent process can block on any single request without locking out further requests, either from another process on the same client system or from another client system altogether. If a second request comes in before the first one completes, a second agent process is found from the process pool to perform the new request.

8.3 Mount Points

A tricky problem in providing a fully transparent EFS is the correct handling of file system "back pointers" (i.e., links to shallower levels of the directory tree). Under UNIX, every directory initially contains two entries: "." (pronounced dot) and ".." (pronounced dot dot). The file "." is a link to the directory itself. This is provided as a convenience for the user. The file ".." on the other hand, is a link to the parent directory of the current directory. Thus if the user types:

```
cd foo
[any set of commands that don't
change the current directory]
cd ..
```

The user first enters the directory "foo," performs the given set of commands and then when the command "cd .." is executed, the user is placed in the directory from whence he originated. A problem arises if the directory "foo" is actually a remote directory. After the user executes the *chdir(2)* system call that moves his current directory to the remote machine, all pathnames are relative to the remote directory. Yet the directory in which the user lands on the remote machine contains an entry called: "..". When the second *chdir(2)* call is executed to perform the "cd ..", the server really needs to gate the user back to local host; *not walk back through the path on that machine pointed too by "..".*

This example demonstrates the need for the concept of a "mount point." When a file system is mounted remotely, the remote system must mark the in-core *inode* for that directory as a "mount point" for a remote file system. When a *namei* path walk is run and a "mount point" is crossed, the remote must decide if the processes that is walking this path is an EFS agent process or a normal user process. If it is a normal user process running on the remote host, the "mount point" is ignored and ".." is followed. However, if it is an agent process that is walking the path and the "mount point" originated from the client on whose behalf the agent is acting, *namei* must return a message back to the client system stating that the remainder of the path must be interpreted locally.

8.4 The *stat*(2) problem

In the UNIX Timesharing System, a system call is provided to get information about any file in the system. The call, *stat*(2), returns many pieces of information about the file. Two of those pieces are the *I-number* and the *device number* on which the file is stored. This information can be used by user programs such as the UNIX file copy utility, *cp*(1). One of the better human engineered features of *cp*(1) is that is does a *stat*(2) call on the source and the destination. It the compares the two *device numbers* and *I-numbers* for equality. If they are the same, *cp*(1) recognizes that the user has requested it to copy a file onto itself which would destroy the file, so it returns an error. Unfortunately, under EFS a file is no longer uniquely identified by just its *device number* and *I-number*. The comparison must now include a machine identifier. In EFS, the unique *network handle*, a 32 bit number, is returned as a new field in the *stat*(2) structure that contains the machine identifier.

Thus the *cp*(1) program had to be changed to use the new information. It now tests for equality of all three pieces that identify each file: *device number*, *I-number*, and the *machine indentifier*. To continue to meet the constraint that old code does not have to be recompiled, we retained the old *stat*(2) system call with its original system call number, but renamed it to *oldstat*(2). We then introduced a new *stat*(2) system call with a hidden third argument.[7] This

7. The third argument is maintained by the C library linkage. The user never sees it.

argument is the number of bytes that the *stat*(2) system call is expecting. As a result, all code that is compiled and linked will use the new call, as the old call is no longer in the distributed C library.

In this manner, we continued to have binary compatibility between the old user binaries and the new operating system product, but have warned users that a certain class of programs, that were written without EFS in mind may exhibit incorrect behavior when used across two different machines. In the case of *cp*(1), unless we had fixed it, it would have returned an error to a user stating that two files were the same file when in fact they were not. This behavior would not damage anything, but would cause a great deal of aggravation to the user.

8.5 The *ustat*(2) problem.

Some versions of UNIX contain a *ustat*(2) system call. This call returns information about a given file system, such as the amount of free space on a device that is passed as a parameter. The *ed*(1) text editor, performs a *stat*(2) on a file to see onto which device the file will be stored, and then calls *ustat*(2) to see if there is enough space to store the file. Unfortunately, with EFS the *device number* will be from the host where the file is stored, and *ustat*(2) will be local. *Ed*(1) was not working correctly within the multi-machine environment. The solution was to introduced a new system call, *rustat*(2) that takes the *machine identifier* as a parameter and then change *ed*(1) to use it.

Again, this is a function of a program that was trying to do something that is valid on a single machine, but not valid in a multi-machine environment. It is far easier to fix those few programs that do not work correctly, than it is to try to come up with a hack that fakes them into working.

9. What EFS Cannot Do

9.1 Remote Devices

In its present form, EFS provides transparent remote access only to disk files; it does not support remote devices. The reasons for this are twofold. First, when performing I/O operations on certain devices, *ttys* in particular, there is the possibility for indefinite delays to occur. For example, a read from a user's terminal device may block for hours while he steps out for a bite of lunch. In contrast, accesses to disk files, though they may cause the requesting process to block, always complete within a short period of time. In the EFS environment where a remote access is actually carried out by an agent process on the system where the resource physically resides, it is crucial that the operation be completed quickly. This is to ensure that the agent processes (a relatively scarce resource) do not all become blocked indefinitely, which in turn ensures that a remote client will always receive service in a timely manner.

The second difficulty involved with remote device access is the handling of the *ioctl* system call. The heart of this problem is that an *ioctl* command may call for the transfer of an arbitrary amount of data between the requesting program's address space and the device driver that implements the command. Further, the format of the data and the direction of transfer are completely determined by the device driver; that information is unavailable to any other part of the kernel. In the case where the device driver is on a remote machine, the local system cannot know *a priori* how much data, if any, to fetch from user space to send to the server. An obvious solution is to have the server request the appropriate amount of data from the client as needed, though it does incur the overhead of extra network transactions. Other approaches to this and other problems with remote devices are the subject of future EFS development work.

9.2 Diskless Nodes

The concept of "diskless workstations" is one that has gained increasing popularity of late. EFS in its current state does not directly address this issue. It assumes that each participating workstation has at least some local backing store containing its root file system, operating system image, paging area and various other files and programs necessary to boot up. Obstacles that stand between EFS and a completely diskless environment include remote paging, obtaining the initial boot image, and construction of an in-core root file system. An additional problem that arises where the network interface is an intelligent front-end processor is the initial down-loading that must precede any other network communications.

Though the authors believe that many users of diskless workstations rapidly find themselves in the market for add-in mass storage, this is another area where future development effort will be concentrated.

10. Summary

The principal motivation behind the MASSCOMP EFS project was the desire to offer our users fully transparent access to files on remote computers. The benefits yielded by this capability are more cost-effective use of mass storage capacity through reduced duplication of common files, and simplified management of shared databases. The goal of transparency has been achieved by building the concept of "remoteness" into the existing UNIX file system I/O architecture. In the same way that the existing UNIX *mount* system call attaches a local disk partition into the file system hierarchy, the new *rmount* system call attaches a subtree of a remote system's directory structure into the local name space. Because this functionality is built into the kernel, and not implemented as a user-level library, existing programs may access remote files without recompilation, and indeed without even being aware of their remoteness.

The implementation model is based on client-server interactions. Each file-related system call (as well as several internal system routines) has a client counterpart that takes responsibility for contacting an agent process on the remote system, passing the necessary information to it, and receiving the response. Each system supporting EFS also maintains a pool of kernel processes that perform the operations requested by remote clients. The principal kernel structure that describes a file, the *inode* is augmented with additional information to form an *rinode*. It is the *rinode* that informs a system call of a file's remoteness and provides the key that allows the client to identify the correct file to the server where it physically resides.

Inasmuch as the UNIX file protection scheme is based not on login names but on numeric user-ids, it was felt to be essential that all machines participating in an EFS share a common login-name to user-id correspondence. This is not only to avoid the complication of having to dynamically translate from one mapping to another, but also to promote the overall transparency of the system. Part of the EFS project included developing tools to assist individual system administrators in bringing their local file systems into correspondence with a network-wide mapping.

When EFS was being designed, consideration was given to what kind of communication services were necessary to support it. Because a fair amount of complexity in kernel modifications was anticipated, there was a strong desire to keep the communications interface simple. An examination of the then available protocols (TCP and UDP) revealed that neither was particularly suited to the task. TCP provides reliable connections but requires too much setup overhead to create a new connection for each client-server interaction. UDP is more suited to the client-server transaction model but does not provide for reliable delivery and has no capability to de-multiplex messages directed to different processes without requiring a socket per process. In view of these shortcomings, a reliable datagram protocol (RDP) was designed and implemented, providing guaranteed message delivery (or notification of failure) and the ability to dynamically create multiple low-overhead connections through a single socket.

One of the more challenging aspects of the EFS project was finding solutions to all of the new error situations that can arise in a distributed environment. Mechanisms were developed to detect and recover from errors resulting both from failure of a remote machine and from the disruption of network communications.

Areas not currently addressed by EFS are support for remote device files, and the operation of systems without local mass storage. Both of these areas are of interest to the authors and are the subjects of ongoing development efforts.

11. Credits

The authors would like the thank the members of the MASSCOMP staff who reviewed not only the ideas as they were developed, but the code. Finally, we would like to recognize and thank John Sundman of the MASSCOMP Documentation group who helped review this paper and certainly improved its structure.

Arn84a. Arnovitz, D., *Ideas about Distributed File Systems - Private Communication,* MASSCOMP (July 1984).

Bil83a. Billingsley, Giles and Keller, Ken, "Program Reference for KIC2," *Electronics Research Laboratory Memorandum* (December 1983).

Bro82a. Brownbridge, D.R., Marshall, L.F., and Randell, B., "The Newcastle Connection," *Software Practices and Experiences*, pp.1148-1162, Computing Laboratory - University of Newcastle upon Tyne (July 21, 1982).

Buc84a. Buck(ed), D. A., */usr/group UNIX Standard,* /usr/group (March 1984).

Coh76a. Cohen, Ellis, "Program Reference for SPICE2," *Electronics Research Laboratory Memorandum* **ERL-M592** (June 14, 1976).

Duf82a. Duff, Thomas, *An Extended File System for Unix Version 7,* LucasFilm (1982).

Gre83a. Greenwald, Michael and Allen, Larry W., *Ideas for an Remote Virtual Disk - Private Communication,* CSL - MIT (October 1983).

IEEd)a. IEEE, P1003 Working Group, *IEEE P1003 POSE - Portable Operating System Environment - Draft,* IEEE (January 1986 (expected)).

Kel84a. Keller, Ken, "Program Reference for HAWK," *Electronics Research Laboratory Memorandum* (June 1984).

Lef82a. Leffler, S. J., Fabry, R. S., and et., al., *4.2 BSD System Manual,* Computer Systems Research Group, EECS-UCB (June 1982).

Lef82b. Leffler, S. J., Fabry, R. S., and et., al., *4.2 BSD Interprocess Communication Primer,* Computer Systems Research Group, EECS-UCB (June 1982).

Lyo85a. Lyon, Bob, Sager, Gary, and et., al., *Overview of the Sun Network File System,* Sun Microsystems, Inc., Mountainview, CA (January 1985).

Pos80a. Postel(ed.), Jon, "User Datagram Protocol - UDP," *RFC 768*, USC/Information Sciences Institute (August 1980).

Pos81a. Postel(ed.), Jon, "Transmission Control Protocol - TCP," *RFC 793*, USC/Information Sciences Institute (September 1981).

Pos81b. Postel(ed.), Jon, "Internet Protocol - IP," *RFC 791*, USC/Information Sciences Institute (September 1981).

Pos82a. Postel(ed.), Jon, *Internet Protocol Transition Workbook,* SRI International, Menlo Park, CA (March 1982).

Qua83a. Quarles, Thomas, "SPICE3: User's Manual," *Master Report, EECS - UCB* (1983).

Rit74a. Ritchie, D.M. and Thompson, K., "The Unix Time-Sharing System," *Communications of the ACM* **17(7)**, pp.365-375, The Bell Telephone Laboratories (1974).

Rit78a. Ritchie, Dennis M., "The Unix I/O System," in *Unix Programmers Manual*, The Bell Telephone Laboratories (November 12, 1978).

Wei84a. Weinberger, P.J., "The Version 8 Network File System," *1984 Unix Summer Conference*, AT&T Bell Laboratories (June 1984).

A Debugger for the Unix Kernel

Steven A. Zimmerman

Masscomp
One Technology Park
Westford, MA 01886

ABSTRACT

Kdb is a version of *adb* that lives in the kernel. *Kdb* offers essentially the full power of *adb,* allowing programmers to breakpoint the kernel, examine and modify locations in a live system, and even single step through kernel code. Additionally, the *crash* program has been integrated into *kdb,* so that system tables and buffers may be easily examined during the debugging process.

Introduction

One of the main reasons that kernel development work is usually slow and tedious is that there are rarely adequate debugging tools available. The *adb* program is generally used for this purpose, either on a crash dump or on **/dev/kmem.** Unfortunately, the most useful features of a debugger, such as breakpointing and single stepping, are not available when *adb* is used this way. Additionally, there is no easy way to stop a running kernel, take a look around, and then resume running the kernel and user processes. The alternative of modifying **/dev/kmem** on the fly with *adb* is risky at best, and the programmer is still not provided with the control over the environment that is so often necessary for effective debugging.

For these reasons, the *kdb* kernel debugger was developed at Masscomp. *Kdb* is essentially a version of *adb* that has been adapted to run as part of the kernel. Whenever a new kernel is made, a special option to *make* indicates that *kdb* is to be included in the kernel; otherwise, a dummy file is included to resolve references to *kdb* from other parts of the kernel. *Kdb* is included only as an option because it is large, occupying approximately 130K of memory.

Invoking Kdb

Kdb may be called in a number of ways. It may be invoked at boot time from the command line to the standalone shell as follows:

```
$$ unix -debug
```

In this way, *kdb* is entered just after Unix is initialized. This allows the setting of breakpoints early on. Once Unix has been booted, *kdb* may be invoked from the console by typing **^P** any time the console is not in raw mode. Since **^P** is caught very early on by the console device driver, this method often allows debugging of systems that otherwise seem to be hung. Alternatively, the **kdb** program may be executed by root from any terminal to bring up *kdb* on the console; this is useful if the console itself is not responding. Additionally, *kdb* is always entered when the system panics, right after the panic string is printed out. In practice, we have found

that this feature has meant that crash dumps are no longer necessary after panics. Operating system developers occasionally find it useful to have some branches of their code invoke *kdb* automatically; this can be done by inserting at the appropriate place in their code the line

```
asm("trap 14");
```

Using Kdb

Kdb usage is almost identical to that of *adb*. One noticeable difference is that *kdb* has a prompt in the form of a right angle bracket, which makes it much more obvious when *kdb* is waiting for input. Following is a short sample *kdb* session to set a breakpoint. In this case, *kdb* was entered by means of the **kdb** command.

```
% kdb
Entering debugger.
68K stack frame pointer (A6) = 4eee

KDB Kernel Debugger
> ttread + 50?i
__ttread + 50:          bsr     __canon
> :b
> $b
breakpoints
count   bkpt            command
1       __ttread + 50
> :c
unix: running
%
```

The Symbol Table

Kdb needs to have the kernel symbol table available to it at all times, since it cannot depend on having access to the file system, especially when it has been entered from a panic. For this reason, the symbol table is loaded directly into the kernel when the kernel is first made. A large array is reserved inside *kdb,* and a special utility that is invoked at *make* time finds this array and writes the symbol table into it. This makes the symbol table always available, so that *kdb* does not have to worry about which version of Unix is really being run and where its symbol table really lives.

Breakpoints

Breakpoints and single stepping in *kdb* work in very much the opposite way from the way they work in *adb*. In *adb,* the program to be debugged is run as a subprocess under *adb,* and when a breakpoint is hit or a single step is finished, control passes back up to *adb*. By contrast, *kdb* is run as a kernel subroutine, so that when a kernel breakpoint is hit, the trap routine calls *kdb*. The **:c** (continue) command in *kdb* causes *kdb* to relinquish control, effectively exiting until the next breakpoint or invocation of *kdb*. Breakpoints can be set on a system running multiuser with many processes; when a breakpoint is hit, everything just comes to a halt (including the clock). A kernel can be halted and resumed this way as many times as desired with no loss of data. Since the **:c** command resumes kernel execution, the **$q** (quit) command is used to sync the disks and return the system to the standalone shell.

Implementation

In many ways, making *kdb* work in the kernel was a matter of taking the *adb* code and simplifying it. All the **ptrace** calls were removed; most of them were replaced by simple memory access routines. These routines allow *kdb* to operate in kernel virtual address space, which is what is generally desired; for example, this mode allows references to the current user structure to be interpreted correctly. Additionally, device addresses are recognized and handled properly so that it is possible to examine and modify various device registers. There is one significant limitation here, however; the memory access routines all operate a byte at a time, so that trying to read or write device registers that must be accessed a word at a time will not always work.

Character input in *kdb* was implemented by adding a new *getchar()* routine to the console device driver. Since *kdb* runs at a very low level, normal erase and kill processing is not available, and this code had to be added to *kdb*. *Kdb* is able to look at the console's tty structure, though, and in this way uses whatever erase and kill characters were in effect when it was invoked. If *kdb* is invoked with the **-debug** switch at boot time, then a default set of erase and kill characters are used. An interrupt facility is simulated, so that the user's normal interrupt character causes *kdb* to discard the current line and print a newline and a prompt.

Breakpointing in *kdb* was implemented by having *kdb* plant a **trap** instruction with an unused opcode in the kernel; the *trap()* routine was then modified to look for this special type of trap. Single stepping was implemented by having *kdb* set the trace bit in the saved program status word; again, the *trap()* routine was modified to detect this condition.

Due to its general nature, *kdb* can be used to debug almost any part of the kernel. The only parts of the kernel that cannot be breakpointed are the *kdb* code itself, parts of the *trap()* routine, the kernel printf routines (which are used by *kdb),* and the routines in the console device driver that handles kernel output. These last routines do not use the normal tty driver at all though, so the tty routines can be breakpointed as freely as any other part of the kernel. In fact, Masscomp has just developed a new tty driver, and *kdb* was used extensively in the debugging phase of this project.

Crash

In addition to the normal *adb* commands, *kdb* also contains all the commands from the *crash* utility. These commands tend to be very useful in the context of *kdb* usage, allowing programmers to see how the contents of various system tables and buffers change during kernel execution. *Crash* commands are differentiated from normal *kdb* commands by beginning with a percent sign. For example, the *crash* command to print out the proc table would be

```
) %proc
```

Unlike *kdb,* the memory access routines in *crash* were not all centralized in one place. Instead, each command did its own memory access through a combination of *lseek* and *read* commands on **/dev/kmem,** copying the results into local variables or structures. The porting of these commands was accomplished generally by turning the *lseek* and *read* commands into direct memory accesses, with the results again being stored in local variables. Where convenient, even this step was bypassed, and the *crash* variables were made to point directly to the appropriate kernel variables.

Summary

In the few months that *kdb* has been in use at Masscomp, it has already proven quite valuable, and the operating system developers routinely compile it into their kernel. A good example of the perceived importance of *kdb* at Masscomp is that when Unix was recently ported to a new machine, *kdb* was the first part of the kernel to be brought up.

Multi-process Debugging

Mark Himelstein
MIPS Computer Systems, Inc.

Peter Rowell
Third Eye Software, Inc.

In recent years, programmers have become more sophisticated and more expensive. As a result, interest has increased in better tools to provide more freedom and improved efficiency. Several high level debuggers, (e.g. *sdb* and *dbx*) are available in the UNIX™ world because of this demand for better tools. None of these debuggers, however, have addressed an issue integral to UNIX systems programming: programs that use *fork* and *exec*.

Debugging was originally accomplished by the use of assembly level debuggers or by the inclusion of *printf* statements in the code. Several source level debuggers have improved on this for programmers working with single image, non-forking processes. UNIX systems programmers are still using the original techniques for lack of a sufficiently powerful tool. One debugger, CDB™, provides a robust, source and assembly level debugging environment for both single *and* multiple processes.

CDB facilitates development by providing extensive flow control, flexible program state examination and manipulation capabilities, a full C expression evaluator and remote debugging. The flow control commands include attributed breakpoints and assertions, continues, goto's, conditionals and single steps. CDB allows the user to analyze program errors and test possible solutions, without recompiling, by coupling the aforementioned capabilities with full read/write access to the call stack, local and global variables, and machine registers. CDB also allows remote debugging across networks, which is useful for stand alone programs, kernel work and other situations precluding the use of a full UNIX context. Lastly, CDB provides the same capabilities for multi-process, multi-image programs.

Why hasn't "multi-process" been done before? In the past, debuggers have not addressed this issue for several reasons:

— General debugging technology was still in its youth,

— Very few programmers needed the capability,

— Programmers were thankful for *any* help and were therefore less demanding.

Programs, and the architectures they run on, have become more sophisticated. This has provided a demand, justification and impetus the development of multi-process debugging tools.

Two areas in particular are discussed in this paper — the user interface and the multi-process implementation. The user interface must provide a context-switchable environment with process granularity and presents the problems of incorporating and integrating the multi-process concept into a previously single process domain. In fact, encapsulating the process structure in the debugger and presenting a coherent picture to the user constituted a majority of the labor spent in completing multi-process CDB.

CDB's concepts of *current file*, *current procedure* and *current line* have been extended to include the concept of *current process*. All commands that examine, modify or in any way affect a

---

CDB is a trademark of Third Eye Software, Inc. UNIX is a trademark of AT&T.

process apply only to the current process. The following is a summary of CDB's process oriented commands:

list process — This command lists the existing debuggable processes. Each process is assigned an ordinal number for reference in other process commands. The current process is marked with a '〉'. For example:

```
  #    PID  Status     SOE BPs Name and Arguments
  0  10318  Running     0   1  some_program
〉 1  10320  Stopped     1   1  some_other_program
```

This shows the CDB assigned process number, the UNIX process id, the current status, whether the process (and any children) should stop after an *exec*, whether breakpoints should be copied after a fork, and the program name and arguments for the process.

process # — Change the *current process* to process #. All other commands assume the new process as the default.

process signal N — Send *signal* number N to the current process. In a UNIX environment, a DEL (or ^C or whatever) will cause a signal SIGINT to be sent to *all* processes descended from CDB.

process exec — Toggle the *Stop-On-Exec* flag. This allows you to grab control before a program takes off after an exec.

When a program forks, the kernel makes a copy of the process image from the existing process. If there are any breakpoints set in the parent before a fork, both parent and child will contain them after the fork. When a program execs, the kernel loads a new image instead of the one attached to the process at the time of the exec. Therefore, the new image contains no breakpoints and if CDB allows the process to just continue execution, the user could never gain control over the process in an orderly way. If this flag is on, a process will act as if it received a temporary breakpoint at the first executable line of a new image. This enables users to get control of a program after it execs. Children of forks inherit this flag in their process descriptors.

create program_name [-d directory ...] — Create a new process, which will be a direct child of CDB. You may optionally specify alternate directories where the executable and/or the sources for it may be found.

kill # — Kill process number #. If # = = '*', kill everything in sight.

The single step command is affected by multiple processes in an expected way.

— If you single step over a *fork*, both the parent and child will stop after the fork.

— If you single step over an *exec*, the *Stop-On-Exec* flag is turned on temporarily until the exec'ed process stops.

One last note on the user's interaction with multi-process CDB: it is generally asynchronous. At any time, it may handle signals or breakpoints for *any* running, debuggable process as well as handling user requests from the terminal. Additionally, one may receive output from an active process while typing in a CDB command. This may be compared to the process status reports provided by *csh*, in the Berkeley release, when the *notify* flag is set. We are currently exploring ways of reducing the potential confusion caused by these events.

The actual implementation of multi-process debugging is constrained by using only unmodified vendor software to accomplish a task that has not been done before in the UNIX world. The solution is as simple as it is intriguing. CDB accomplishes multi-process debugging by exploiting existing communication mechanisms and using debug servers to control the actual processes being debugged.

Why is multi-process debugging under UNIX different from single process debugging? To regress, a process can fork a child and then monitor that child using the *ptrace* system call, but *only direct parents* can use this facility. In the single process case, CDB forks the process to be debugged and is thus the direct parent. Unfortunately, *ptrace*'s restrictions will not allow CDB to use the same mechanism to debug a non-immediate, descendant process. Because of this restriction, programs that fork present a problem. CDB handles multiple processes by having the debugged process fork a debug server (who's code is included via a library) which in turn forks the actual child. This plan assumes two things: the debug server can 1) control the child process in some manner and 2) communicate with the original invocation of the CDB debugger, which monitors its actions. CDB satisfies the former by using the *ptrace* and *signal* facilities and the latter by using BSD 4.2's TCP/IP. The implementation is not strongly wedded to TCP/IP and can be made to work in other environments supplying sufficient communications flexibility. In fact, this method is almost identical to the one used by CDB for remote debugging.

The *exec* system call is another major area of concern — when a process changes its image, how do we track it? Having established the capability of debug servers communicating with CDB, we can use the same facility in this situation. When a process decides to *exec*, it first notifies CDB, *then* it execs. In this way CDB can maintain the correct information regarding the debugged process.

For single process debugging, the user may optionally load an extra module to help CDB with certain functions. This module has been extended for multi-process CDB by including replacements for *execv* and *fork*. These replacements contain the code that forks debug servers and maintains communication with CDB. On the all of the systems we have examined, all variants of *exec* ultimately call *execv*. By replacing execv and having our version use the *syscall* mechanism, we effectively trap all execs in a given program. Fork is more difficult because you cannot use syscall on it. For this reason, we require users to define *fork* to *FORK* with the C preprocessor facility causing the user program to call our procedure FORK which in turn calls the system's fork. Most vfork's can be debugged by defining *vfork* to *FORK* in the same manner as fork.

Multi-process CDB solves a large set of multi-process and multi-image problems. Is is possible, however, to do more. It should be possible to connect a debugger to an actively running process. It is also desirable to debug the "production" version of a multi-process system, i.e. do away with the necessary recompile and load. These goals can be achieved by enhancing existing system capabilities in the following way:

- Ptrace is modified to not require a strict parent-child relationship. Ideally, no implicit relationship exists between the processes other than, perhaps, ownership by a given user.
- A new ptrace request (10) is added. This is identical to request 0, except that the pid of the target process is specified. This allows us to grab an already running process and start debugging it. (Like that hung *tar* process you had last week.)
- When a process has its internal trace bit on (STRC), *exec* and *fork* cause the signals SIGEXEC and SIGFORK to be generated. This permits the debugger to deal with these events. It is also necessary for the debugger to obtain additional information, such as the parent pid for the fork, and the new program name for the exec.

- Additional modifications to ptrace would improve overall efficiency. For example, block moves to and from the child's registers or the child's memory image would both speed up the debugger's interaction with the process and reduce the general system load by minimizing spurious context switches.

In conclusion, CDB provides UNIX programmers with a major new capability: multi-process debugging. This facility will shorten development times and increase programmers' creativity. CDB finally allows programmers to use a debugger to debug *all* UNIX programs.

The Fourth Generation Make

Glenn S. Fowler

AT&T Bell Laboratories
Room 5D-105
Murray Hill, N. J. 07974
ulysses!gsf

ABSTRACT

Make is a program that is widely used to maintain and update programs and libraries on UNIX* systems. This paper introduces the *Fourth Generation Make* which embodies major semantic and syntactic enhancements to the standard *make* program. The enhancements include support for source files distributed among many directories, an efficient *shell* interface that allows concurrent execution of update commands, dynamic dependency generation, dependencies on conditional compilation symbols and a powerful new meta language for constructing default rules. A complete rewrite of the standard make code has resulted in a unified software construction program that also provides improved functionality and performance. It is assumed that the reader is familiar with the features and operation of the standard *make*.

1. INTRODUCTION

Since it first appeared in 1976[1], *make* has changed little, and most of these changes have been syntactic in nature[2]. However, the *build* variant of *make*[3] introduced a major semantic change. *Build* has *viewpaths* that allow source files to reside in more than one directory. *Viewpaths* support the sharing of source files between many users and help eliminate the proliferation of source copies. The semantics of multiple directories, however, conflict with the flat, single directory approach of old-make (*old-make* refers to the *make* distributed with the System 5 UNIX system and *new-make* refers to the Fourth Generation *make*). *Build* must *link* or *copy* files into the current directory to comply with the single directory nature of *old-make*. Without *build*, multiple directories are usually handled by using recursive *makes*. Because of the large data area (which must be copied on *fork*) and the time consumed by the startup mechanisms (parsing builtin rules, scanning directories, etc.), recursive *make* calls are inefficient.

The Fourth Generation *make* is the result of a research effort to improve the execution and semantics of *old-make* and to update the model of *make* to be more consistent with such concepts as multiple source directories and concurrent execution of update commands.

The main features of the Fourth Generation *make* are that it:

- Efficiently supports both source and targets in multiple directories.
- Provides a more efficient interface to the *shell*.
- Decreases redundancies inherent in old makefiles.
- Refines the granularity from the *file* to the *variable* level.
- Provides mechanisms for writing portable makefiles.

---

* UNIX is a trademark of AT&T Bell Laboratories.

- Provides a powerful metalanguage for constructing builtin rules.
- Supports program configuration and generation for many UNIX system environments.
- Compiles makefiles into binary form for reduced startup time.
- Provides an abbreviated syntax to accommodate the most common uses of *make*.
- Provides optional dynamic dependency generation.
- Provides a natural interface for object library maintenance.
- Provides a natural interface to SCCS.

This paper describes the motivation and design of the new features of the Fourth Generation *make* and also discusses the advantages of using *new-make*. Some of these advantages are:

- **Decreased makefile size** — most *new* makefiles consist of one or two lines of *source* file dependencies. Conversion of the BSD and 3B2 kernel makefiles shows a factor of ten decrease in makefile size.
- **Consolidation of makefiles** — usually, recursive configurations of old makefiles can be replaced by one makefile. This occurred in the conversion of BSD and 3B2 kernel makefiles.
- **Improved consistency checking** — duplicate source and header files are reported. Additionally, missing header files are reported before compilation takes place.
- **User definable builtin rules** — the default builtin rules are just a compiled makefile binary, so there is no loss of efficiency when other default rules are used. The default rules can either be completely bypassed or augmented, depending on the input options to *make*.
- **Decreased execution time** — experiments on large makefiles (when all targets are up to date) show a factor of ten average decrease in execution time. The shell interface also eliminates one *fork* and one *exec* for each makefile command line executed. In contrast to *old-make*, which *forks* for each command line, *new-make forks* only once to initialize the *shell* co-process. After this initialization the shell is the only *forked* process.
- **Efficient file sharing** — software development teams can easily share single copies of source and object files.
- **Improved software organization** — the multiple directory nature of *new-make* encourages the organization of large software project files into more than one directory. Unlike *build*, *new-make* allows files to be distributed over arbitrary directory structures.

Since the full functionality of *new-make* cannot be realized using the restricted model provided by *old-make*, *new-make* does not support old makefiles. This is because the research effort of *new-make* emphasizes increased execution speed and simplicity of use over issues of backward compatibility.

The remaining sections elaborate on the features and improvements introduced in *new-make*. This paper is not a tutorial and so does not contain a detailed description of specific command arguments, options or actions. All timing estimates are based on the user and sys times of programs running on a VAX/750*.

---

* VAX is a trademark of Digital Equipment Corporation.

2. MAKEFILES

Makefiles are typically at the source of software portability problems. More often than not, makefiles are run in combination with shell scripts to determine the current environment parameters. In the worst cases, the user is interactively prompted for the desired information.

A major goal of *new-make* is to encapsulate as much information as possible into a single makefile. A new shell interface and makefile preprocessor make this encapsulation possible. As a result it is possible for the same makefile to run on different machines and even on different versions of the UNIX system.

2.1 Preprocessing

Each makefile is preprocessed by the C preprocessor *cpp*, making the full flexibility of include files and macro definitions available to makefile users. The version of *cpp* distributed with the *new-make* also contains some changes that improve makefile portability. **#if** statements may use the following builtin tests:

exists(*file*)
: Returns 1 if *file* can be found using the **#include** search rules. *File* may optionally be enclosed in ⟨ ⟩, just as in the **#include** statement. This test can be used to tailor software generation for specific UNIX variants and environments.

identifiers(*file*,*id1*,...)
: Searches *file* for the null terminated identifiers *id1*... and returns the number of identifiers found. Initial __ characters in both *id1*... and identifiers in *file* are ignored. If *file* is an archive with a symbol directory then only the symbol directory is searched. Therefore, this test can be used to determine if a function or global symbol is present in an object library.

Additional function predicates are defined by **#assert** statements in the include file ⟨**default.h**⟩ which is automatically included by *cpp* before the first input file is read. **#assert** *predicate*(*value*) makes assertions that can be tested in **#if** expressions. Such assertions are only recognized within **#if** expressions and do not conflict with **#define** variable expansions. Most assertions deal with the current machine environment:

system(*system-name*)
: Defines the operating system name. Example values for *system-name* are **unix** and **gcos**.

release(*system-release*)
: Defines the operating system release name. Example values for *system-release* are **apollo, bsd, research, sun, system5, uts,** and **venix**.

version(*release-version*)
: Defines the operating system release version. Example values for *release-version* are **4.1c** and **4.2** for **release(bsd), 7** and **8** for **release(research)** and **5.0** etc. for **release(system5).**

model(*model-name*)
: Defines the hardware model name that also encompasses workstation names. Example values for *model-name* are **apollo, sun, ibm-pc** and **unix-pc.**

architecture(*architecture-name*)
: Defines the processor architecture name. Example values for *architecture-name* are **3b, 68000, ibm, pdp11,** and **vax.**

machine(*architecture-version*)

Defines the processor architecture version. Example values for *architecture-version* are **2**, **20** and **20s** for **architecture(3b)**, **70** etc. for **architecture(pdp11)** and **750**, **780** and **micro** for **architecture(vax)**.

2.2 Compilation

When all targets are up to date, *old-make* spends most of its time parsing the builtin rules and input makefiles, whereas *new-make* avoids this problem by compiling makefiles into binary make object files. The input makefile *x*.**mk** is automatically compiled into the make object file *x*.**mo** whenever the object file is out of date with the corresponding makefile.

Compilation takes place after the makefiles are read and parsed and involves writing the internal structures to a file in relocatable form. The time spent compiling is negligible compared to the parsing time and loading compiled makefiles saves an average of 2 seconds on program startup.

Whereas *old-make* must parse its builtin rules on program startup, *new-make* loads the standard builtin rules from a compiled makefile. Since the standard builtin rules are placed in a compiled makefile, substituting local builtin rules for the standard ones results in no loss of performance. Local builtin rules can be specified either on the *make* command line or in the **MAKERULES** environment variable.

2.3 Operators

New-make supports (two character) operators that may be defined in the builtin rules file. The usage syntax of these operators is exactly like the : dependency operator, but the semantics depend on the specific operator definitions.

Only one new operator, ::, is provided in the standard builtin rules. Unlike the : dependency operator, which typically specifies *object* file dependencies, :: may be used to specify *source* file dependencies:

```
program : p1.o p2.o p3.o
        $(CC) -o program p1.o p2.o p3.o -lm
```

can be specified as

```
program :: program.mk p1.y p2.l p3.c -lm
```

Using :: eliminates the duplication of file name lists and also provides other important features (see **Dependency Generation** and **Common Actions** below).

3. BINDING

A basic internal operation is the *binding* of rule names to objects (e.g., files and variables) in the system. *Binding* enables files to be placed in more than one directory and also allows targets to depend on variable definitions as well as files.

3.1 Files

Rule names are bound to file names using the dependencies of the special **.SOURCE** and **.SOURCE**.*x* rules. The dependencies of these rules are directories to be scanned when searching for files. The current directory is always scanned first. Then, files with suffix .*x* are searched for in the directories specified by **.SOURCE**.*x*. Finally, the **.SOURCE** directories are searched. The left to right **.SOURCE** dependency ordering is important; *make* warns when a file is found in more than one directory, but continues with the first file found.

3.2 State Variables

A *state variable* is a variable whose modify time and definition is stored from one invocation of *make* to the next. *State variables* have two basic forms: *(variable)* is a makefile or *environment* variable and *file(variable)* is a variable corresponding to a **#define** statement in *file*. For example:

```
x.o : header.h(DEBUG) (MACHINE)
```

specifies that *x.o* depends on the definition of the variable **DEBUG** in the file *header.h* and the definition of **MACHINE** from the current makefile. If either definition changes from one invocation to the next then *x.o* will become out of date and will be regenerated.

A file is scanned for **#define** definitions only if it has been modified since the last time it was scanned. The first **#define** definition for a variable is used and all other definitions are ignored.

The state variable definitions are stored in the state file *base*.**ms** where *base* is the base name of the first makefile in the argument list. The state file is implemented as a compiled make object file, resulting in little maintenance overhead. Some dependency information is also stored in the state file (see **Dependency Generation** below).

4. VARIABLE EXPANSION

The basic actions of *new-make* are controlled by the builtin rules, with most of the expressive power concentrated in variable definitions and expansions. The addition of operators and editing to variable expansions has resulted in a powerful makefile metalanguage. As a testimony to the strength of this metalanguage, most new *make* features and ideas have resulted in changes to the standard builtin rules (written as a makefile) rather than in changes to the *make* C source files.

The metalanguage produces a more complex set of builtin rules, but in turn allows simple, concise and easy to maintain user makefiles.

4.1 Assignment

There are three variable definition operators:

variable = *value* — *Value* is assigned to *variable* without expansion.

variable := *value* — *Value* is expanded and then assigned to *variable*. This allows a variable to contain all or portions of its previous value.

variable += *value* — *Value* is expanded and appended to the current value of *variable*. This allows lists to be generated in variable values.

4.2 Expansion

Each occurrence of **$**(*variable*) in a makefile is replaced by the assigned *value* of *variable*. The substitution is recursive in that *value* is checked for other variable expansions before being substituted. **$**(*variable*:*operator*) causes *value* to be edited according to the specifications in *operator* before being substituted. : is used to separate multiple *operators*. The general form for *operator* is *op*[=*arg*] where *op* is a single character operator name and *arg* is an optional operator argument. The operators are applied to each space separated token in the expanded variable value. The variable name itself is expanded before the value is determined, implementing primitive variable subscripting.

4.2.1 File Components Because of the multiple directory nature of *new-make* it is important to be able to separate a file name into its individual components. File names are divided into the following five basic components:

M *machine* — All characters up to and including the last !. *Null* if no ! appears. The *machine* component is supported but not used in the current implementation.

D *directory*	All characters after the last ! up to and including the last /. *Null* if no / appears.
P *prefix*	All characters after the last / up to and including the first .. *Null* if there are less than two .'s or if . is the first character.
B *base*	All characters after the first . up to but not including the last ..
S *suffix*	All characters from the last . to the end. *Null* if no . appears.

Here is an example using file name component editing:

```
FILES = a.y dir/s.x.c bozo!.profile

$(FILES)              ->    a.y dir/s.x.c bozo!.profile
$(FILES:B:S=.o)       ->    a.o x.o .profile.o
$(FILES:DBS)          ->    a.y dir/x.c .profile
$(FILES:M)            ->    bozo!
```

4.2.2 Matching Space separated tokens in variable values can be matched using the shell file pattern matching characters in the **:N=***pattern*: **:N!=***pattern*: edit operators. The matching occurs before component editing takes place. Using **FILES** as an example:

```
$(FILES:N=*.c)          ->    dir/x.c
$(FILES:N!=*.c)         ->    a.y bozo!.profile
$(FILES:N=*.[cy]:BS=.o) ->    a.o x.o
```

4.2.3 Substitution Tokens in variable values can be substituted using the **:C**⟨*del*⟩*old*⟨*del*⟩*new*⟨*del*⟩**:** substitute edit command. ⟨*del*⟩ may be any character and **C/** may be abbreviated by /. For example:

```
$(FILES:/ /\:/)         ->    a.y:dir/x.c:bozo!.profile
$(FILES:C%/%--%)        ->    a.y dir--x.c bozo!.profile
```

Notice that the : must be escaped to distinguish it from the : edit operator.

4.2.4 Binding Variable value tokens may also be *bound* and selected by type using the **:T=***type*: edit operator. The types are:

A	Each token that can be bound to an *archive* is expanded.
D	Each token that can be bound to a *state variable* is expanded using the *state variable* definition. The expanded definitions may be used as arguments to the *cc* command.
F	Each token that can be *bound* to a file is expanded using the *bound* file name.
N	If *variable* has a null value then the null string is expanded, otherwise # is expanded. This can be used to specify conditional makefile input.
O	Each token that is bound neither to a file nor to a *state variable* is expanded.
S	Each token that can be *bound* to a *state variable* is expanded.
V	If *variable* has a non-null value then the null string is expanded, otherwise # is expanded.

For example:

```
FILES = x.c (DEBUG) (TEST) libc.a
TEST =
DEBUG = 1

$(FILES:T=D)            ->      -DDEBUG
$(FILES:T=F)            ->      x.c /lib/libc.a
$(FILES:T=S)            ->      (DEBUG) (TEST)
$(TEST:T=V)             ->      #
$(FILES:T=V)            ->
```

5. *RULE ATTRIBUTES*

New-make uses rule attributes to control the disposition of rules and targets. For example, the **.ARCHIVE** attribute specifies that a bound rule is to be treated as an object file archive. This attribute allows the following concise archive dependency specification:

```
lib.a :: a.c b.c c.y x.s
```

Attributes also allow *new-make* to assume some system dependent maintenance responsibilities. For example, on Berkeley variants of the UNIX system, **.ARCHIVE** targets are automatically updated using *ranlib*.

Some attributes are assigned automatically, some are determined by the suffix of the current target (**.a** files are given the **.ARCHIVE** attribute) and others may be assigned explicitly in the makefile.

Another attribute **.USE** allows a single command update sequence to be shared by many targets. For example:

```
a.o : (DEBUG)
        $(CC) $(CCFLAGS) -S $(>)
        $(FIXUP) $(>):BS=.s)
        $(CC) -c $(>):BS=.s)
        $(RM) $(RMFLAGS) $(>):BS=.s)

b.o : (PROFILE)
        $(CC) $(CCFLAGS) -S $(>)
        $(FIXUP) $(>):BS=.s)
        $(CC) -c $(>):BS=.s)
        $(RM) $(RMFLAGS) $(>):BS=.s)
```

can be replaced by

```
a.o : .FIXUP (DEBUG)
b.o : .FIXUP (PROFILE)

.FIXUP : .USE
        $(CC) $(CCFLAGS) -S $(>)
        $(FIXUP) $(>):BS=.s)
        $(CC) -c $(>):BS=.s)
        $(RM) $(RMFLAGS) $(>):BS=.s)
```

A brief description of the attributes and special rules appears in Appendix B.

6. DEPENDENCY GENERATION

The main task of *make* is to verify that any changes made to source files are reflected in the corresponding object files. This verification, however, relies on the proper dependencies being placed in the controlling makefiles. A single omitted dependency can cause inconsistencies between the source and object files.

New-make eliminates some of these inconsistencies by dynamically generating file dependencies from a given set of source files. Such dependencies are automatically generated when the **::** dependency operator is used. The generated dependencies are stored in the state file along with the state variable definitions.

The file dependencies are determined by scanning the files for **#include** statements. A file is only scanned if it has been modified since the last time it was scanned. In the worst cases, automatic dependency generation only doubles the execution time of *make* (not including the time spent updating the targets).

#include dependencies within conditional **#if** constructs are given the **.DONTCARE** attribute that allows *make* to continue if the corresponding files cannot be found.

A minor drawback is that the files depend on *all* **#include** dependencies, even if some of the dependencies are from "compiled out" parts of the source files. However, for a given application the "compiled out" dependencies are rarely modified. A proposed alternative is to use *cpp* to produce the exact dependencies, and feasibility experiments are now underway. Initial results show that *cpp* scanning may be appropriate only after major changes to the source files, and that the basic **#include** scanning is sufficient for most applications.

In any event, even the basic dynamic **#include** file dependency generation provides a more consistent environment than statically generated dependencies using *old-make*.

7. SHELL INTERFACE

The execution of update commands has undergone major performance and semantic changes in *new-make*. All update commands are sent to a single copy of the shell, keeping the shell environment intact between command executions. This includes the effects of *cd* and shell parameter assignments.

Since only one copy of the shell is used, *new-make forks* just once to initialize the shell as a *co-process*. While commands are being executed by the shell, *new-make* continues by checking the dependencies of the next target. Thus the next update command is almost always determined by the time the current command completes.

An added advantage is that command *aliasing* and shell *functions* are preserved in update command blocks. *Old-make* uses either the *system* or *exec* call to execute each command line. In *old-make*, if a line contains shell metacharacters (**$ ¦ ()〉〈**) then it is sent to the shell via *system*, otherwise the command is executed via a *fork* and *exec*. *Aliases* in the latter case are ignored by *old-make*.

7.1 Command Blocks

The new makefile structure supports natural shell command specification. Update command lines for a particular target are sent to the shell as a block, allowing shell **case**, **for**, **if** and **while** constructs to cross **newline** boundaries without intervening **backslash** and **semicolon** characters.

With this co-process arrangement makefiles can be viewed as *labeled* shell scripts. The labels (targets) merely determine when the corresponding command blocks are executed.

To avoid confusion between the use of **$** in *new-make* and the shell, only the **$(...)** forms are expanded by *new-make* (**$$(...)** expands to **$(...)**). **$** in any other context is passed untouched to the shell.

7.2 Communication

Pipes are used for communication between *new-make* and the shell. *Make* sends commands to the shell on one *pipe* and receives status information from the shell on a second *pipe*. The status information is organized into four message packets:

error exit-code
: Sent when a command returns a non-zero exit code.

exit
: Sent when a command block completes.

read make-command
: Sends *make-command* back to *make* to be parsed as if it originated in a makefile. This allows dynamic makefile generation, but this feature has not been used (or recommended) in any major applications.

start process-id
: This packet associates a command block with a subshell process id when concurrent execution is enabled. This allows *make* to wait for its children for proper time accounting.

It is sometimes desirable for *make* to continue with other targets even after an error has occurred. A *trap* on command error was added to the shell to handle this case in a co-process environment. In addition, since entire command blocks are sent to the shell, the **—x** execution trace flag of the shell is used to echo each individual command as it is executed. *New-make* places control commands between the command update blocks and constantly switches between the **—x** and **+x** options. So as not to clutter the execution trace with **set +x** commands, the shell suppresses the execution trace of **set +x** when **set —x** is in effect. Many thanks to Dave Korn for adding these two features to *ksh*[4]. Although *new-make* works best with KSH, it also runs with the Bourne shell[5].

7.3 Concurrent Execution

With the shell as a co-process it is a trivial matter to add concurrent command execution to *new-make*. Internally each command block is simply enclosed by { and }& and a *jobs* table is used to associate targets with process ids. The dependency graph specified by the input makefiles is then used to determine when *new-make* must wait for certain targets to complete.

The **—j***n* command line option is used to specify the maximum number of concurrent jobs. By default only one job is used, but if *n* is greater than 1 then each update command block is sent to a new subshell (background shell). Background shells inherit the environment of the main shell (foreground shell) and the foreground shell inherits the environment of *new-make*. It is important to note that no makefile changes are necessary to support concurrent execution.

Concurrent execution should have a major effect on multi—processor machines and on systems equipped with compiler "black boxes." On such machines it would be possible for each command block to execute on different processors.

8. OPTION GENERATION

The builtin rules automatically generate the proper **—D**, **—I** and **—L** options of *cc* in the **$(CCFLAGS)** variable. The **—D** options are generated from *state variable* dependencies, the **—I** options are generated from the dependencies of the **.SOURCE.h** rule and the **—L** options are generated from the dependencies of the **.SOURCE.a** rule. *State variable* dependencies specified using the **::** operator apply to all dependencies of the corresponding target (global dependencies), otherwise the dependencies apply only to the individual target of each **:** operator.

9. COMMON ACTIONS

When the :: operator is used several common action targets are automatically defined. The common action target *xxx* is defined as *.XXX* in the builtin rules. If *xxx* appears as a command line target and *xxx* has not been defined by the input *makefiles* then the target *.XXX* is made. The common actions are:

clean Deletes all object files corresponding to the current makefile.

clobber Executes the **clean** action and also deletes the target(s) corresponding to the current makefile.

cpio Creates a *cpio* archive of the source files listed after each :: operator. The archive is placed in the file *main*.**cpio** where *main* is the base name of the main target rule.

install Makes the main target and copies it to the directory **$(INSTALLDIR)**. By default, **$(INSTALLDIR)** is **$(ROOT ¦ HOME)/bin** (use **$(ROOT)** if it is defined otherwise use **$(HOME)**) for executable targets and **$(ROOT ¦ HOME)/lib** for object archive targets. The commands associated with the rule **.DOINSTALL** are used to do the copy.

lint Runs *lint* on the input source files. Notice that only global **—D** and **—I** options are passed to *lint* (see **Option Generation**). Any **.l** and **.y** source files are automatically preprocessed if necessary.

print The source files are printed by passing them through the filter **$(PR)** and listing them with **$(LP)**.

tar Creates a *tar* archive of the source files listed after each :: operator. The archive is placed in the file *main*.**tar** where *main* is the base name of the main target rule.

ucpio Same as **cpio** except that only those source files modified since the last **ucpio** are archived. If **$(UTIME)** is defined then it is taken to be a file name whose modify time is used to determine the files to be archived; only those files newer than this modify time are archived.

uprint Same as **print** except that only those source files modified since the last **uprint** are printed.

utar Same as **tar** except that only those source files modified since the last **utar** are archived.

10. SCCS INTERFACE

The SCCS[6] interface is enhanced by the addition of *prefix* rules. The prefix rule *p.* specifies how the file **x** is to be generated from the file *p.***x**. The following rules provide complete support for SCCS files:

```
.PREFIXES : s.

s. :
        $(GET) $(GETFLAGS) $(>) > $(<)
        ...
        $(UNGET) $(<)
```

The ... separates the update blocks into *pre*—commands and *post*—commands. The *pre*—commands are executed to update the target, whereas the *post*—commands are stacked (first in first out) until the last target has been updated.

This implementation eliminates the proliferation of ~ rules found in *old-make* and *build*.

11. CONCLUSION

The Fourth Generation *make* is being used successfully by a growing community of users. The program performs well enough that users are not tempted to bypass *make* by manually issuing compile and *touch* commands.

After nearly three months of constant use (and abuse) the program has settled into a "panic" free steady state and is ready for wider distribution. The features and performance gains of this program should make it an attractive software construction tool for both individual users and large projects. The average tenfold decrease in makefile size should be of particular interest to large software project managers.

```
#
# old makefile
#

DEBUG = -DDEBUG
MAX = 123

CFLAGS = -O -I$(HOME)/include $(DEBUG)

CFILES = dir/a.c b.c c.c d.c
HFILES = c.h
OFILES = a.o b.o c.o d.o

command : $(OFILES)
        $(CC) -o command $(OFILES) $(HOME)/lib/lib.a -lm

a.o : dir/a.c $(HOME)/include/a.h
        $(CC) $(CFLAGS) dir/a.c

b.o : $(HOME)/include/b.h c.h

c.o : c.c
        $(CC) $(CFLAGS) -DMAX=$(MAX) c.c

print :
        pr Makefile $(HFILES) $(CFILES) | lp

lint :
        lint $(CFLAGS) $(CFILES)



/*
 * new makefile
 * all files recompiled if DEBUG value changes
 * c.o recompiled if MAX value changes
 */

DEBUG = 1
MAX = 123

.SOURCE.h : $(HOME)/include
.SOURCE.a : $(HOME)/lib
.SOURCE : dir

command :: Makefile c.h a.c b.c c.c d.c lib.a -lm (DEBUG)

c.o : (MAX)
```

.ARCHIVE

The dependencies of **.ARCHIVE** are suffixes associated with archive files. A file with one of these suffixes is treated as an archive. The **.ARCHIVE** attribute causes the target to be treated as an archive and the **.a** suffix rule is used to update the target. The default archive suffixes are:

```
.ARCHIVE : .a
```

An archive is not updated until all the corresponding object files have been generated. *New-make* will use **.o** files from the current directory if they are up to date, and archive member touching (with the **—t** option) is supported.

.CLEAR

Clears the dependency and command lists for the corresponding target.

.CURRENT

The dependencies of **.CURRENT** are suffixes associated with targets that are only produced in the current directory. The **.CURRENT** attribute marks the target as being produced in the current directory. The default is:

```
.CURRENT : .a .o
```

.DEFAULT

This rule is used as a last resort when no other rules can be inferred to make the current target.

.DONE

This rule is made after all targets have been made and has no affect on the update status of the targets. The commands are always executed in the foreground shell.

.DONTCARE

The **.DONTCARE** attribute causes *new-make* to continue if the target cannot be made. Otherwise, *new-make* issues an error and exits if a target cannot be made.

.FOREGROUND

This attribute causes the target update commands to be executed in the foreground shell. Otherwise the commands may be executed in a background shell. Normally *make* computes future dependencies while update commands are being executed, however, **.FOREGROUND** update commands cause *make* to block until the commands complete.

.IMPLICIT

This attribute causes the implicit suffix rules to be applied even if update commands have been specified for the target. Otherwise the implicit suffix rules are only applied to targets with no explicit update commands.

.INIT

This rule is made before any other target and has no affect on the update status of the targets. The commands are always executed in the foreground shell.

.INSERT

Causes the dependency list of the current target to be inserted rather than appended to the target dependency list.

.INTERRUPT

This rule is made when an interrupt signal is caught and has no affect on the update status of the targets. The commands are always executed in the foreground shell. *make* exits after the commands are executed.

.MAIN

If no targets are explicitly listed on the command line then the first dependency of **.MAIN** is used as the main target. Otherwise the first target encountered that is not a special rule or inference rule is the main target. The dependency list of **.MAIN** is automatically set to be the main target.

.MAKE

The **.MAKE** attribute causes the command update list to be parsed by *make* instead of executed by the shell. Such command lists are always executed, even with the **—n** option on.

.MAKEFILES

Specifies the default *makefile* names. If no explicit *makefile* is specified on the command line then these files are tried in order. The default is:

```
.MAKEFILES : makefile Makefile
```

.MAKEINIT

The commands associated with **.MAKEINIT** are executed by the builtin rules before **.INIT** is made. The standard builtin rules rely on this feature.

.NOTOUCH

This attribute causes the target modify time to remain untouched, even if the corresponding update commands are executed. This allows initialization sequences to be specified for individual rules:

```
main : init header
        echo "executed if header is newer than main"
init : .NOTOUCH
        echo "always executed for main"
```

.NULL

When used as a dependency of a target with no suffix or explicit update commands, causes the commands associated with the null rule " " to be used when updating the target.

.OPERATOR

This attribute marks the target as an *operator* to be applied when reading *makefiles*. Operator names must be exactly two characters long.

.OPTIONS

The dependencies of **.OPTIONS** are treated like command line options. The options take effect immediately when **.OPTIONS** is read.

.PARAMETER

The **.PARAMETER** attribute marks the target as a *parameter* file. A *parameter* file only contains definitions (i.e., **#define** definitions) and comments. The modify time of a *parameter* file is ignored when determining the update status of corresponding targets.

.POST

This attribute causes the target to be made *after* the parent target has been made.

.PRECIOUS

The current target(s) are usually deleted when *make* is interrupted. However, the dependencies of **.PRECIOUS** are not deleted. If **.PRECIOUS** has no dependencies then *all* targets are precious. Targets marked with **.ARCHIVE** are always precious.

.PREFIXES

The dependencies of **.PREFIXES** are file name prefixes of the form *p.* used to infer

implicit rules. The (left to right) prefix order is important; the first inference rule name for which both a file and a rule exist is inferred. The default prefix rules provide a smooth interface to *Source Code Control System* files:

```
.PREFIXES : s.
```

.READONLY

When used as a dependency for **.o** target files, **.READONLY** causes the data portions of the corresponding **.c** source file to be placed into readonly text. **.READONLY** can also be used to place the tables of corresponding **.l** and **.y** source files into readonly text. The commands are tailored to the current host machine. **.READONLY** is implemented as a **.USE** rule and will be obsolete when the **const** data attribute becomes a C language standard.

.SEARCH

The dependencies of **.SEARCH** are suffixes associated with files that are to be searched for implicit file dependencies. The **.SEARCH** attribute marks the target to be searched for implicit file dependencies. When the :: operator is used to specify dependencies, targets with the **.SEARCH** attribute are automatically searched for *#include* header file dependencies. If any of the header files are newer than the dependency file then the corresponding target is updated. The default is:

```
.SEARCH : .c .h .y .l
```

.SOURCE

The dependencies of **.SOURCE** are directories to be scanned when searching for files The (left to right) directory order is important; the first directory containing the file is used. The default is:

```
.SOURCE : .
```

.SOURCE.x

A file with suffix *.x* is searched for in the directories specified by the rule **.SOURCE.***x* before the **.SOURCE** directories are checked. Notice that *.x* must be a dependency of **.SUFFIXES**. The (left to right) directory order is important; the first directory containing the file is used. The default when the **—X** option is off is:

```
.SOURCE.a : /lib /usr/lib
.SOURCE.h : /usr/include
```

.SUFFIXES

The dependencies of **.SUFFIXES** are file name suffixes of the form *.s* used to infer implicit rules. The (left to right) suffix order is important; the first inference rule name for which both a file and a rule exist is inferred. The default suffix list is:

```
.SUFFIXES : .o .c .r .f .y .l .s .sh .h .a
```

.TOUCH

Each dependency of **.TOUCH** is touched as though it has already been made. The dependencies are touched immediately when **.TOUCH** is read.

.UNTOUCH

Each dependency of **.UNTOUCH** is untouched as though it had never been made. The dependencies are untouched immediately when **.UNTOUCH** is read.

.USE

The **.USE** attribute marks the target as a **.USE** rule. Any target having a **.USE** rule as a dependency will be updated using the commands associated with the **.USE** rule.

REFERENCES

1. S. I. Feldman, *Make — A Program for Maintaining Computer Programs*, Software — Practice and Experience, Vol. 9 No. 4, pp. 256-265, April 1979.

2. *Augmented Version of Make*, UNIX System V — Release 2.0 Support Tools Guide, pp. 3.1-3.19, April 1984.

3. V. B. Erickson and J. F. Pellegrin, *Build — A Software Construction Tool*, AT&T Bell Laboratories Technical Journal Vol. 63 No. 6 Part 2, pp. 1049-1059, July-August 1984.

4. D. G. Korn, *KSH — A Shell Programming Language*, USENIX Toronto 1983 Summer Conference Proceedings, pp. 191-202, 1983.

5. S. R. Bourne, *The UNIX Shell*, AT&T Bell Laboratories Technical Journal, Vol. 57 No. 6 Part 2, pp. 1971-1990, July-August 1978.

6. L. E. Bonnani and C. A. Salemi, *Source Code Control System User's Guide*, System V Programmer's Manual.

Array Processing Under UNIX

Peter H. Berens

Apunix Computer Services
1380 Garnet Ave., Suite E-292
San Diego, CA 92109
(619) 484-0074
(UUCP: ...!ucbvax!sdcsvax!dcdwest!phb
or ...!decvax!ittvax!dcdwest!phb)

ABSTRACT

Array processors are fast special purpose computers designed to carry out floating-point computations at very high speeds. They typically are not stand alone computers like the vector processors such as the Cray, but are less expensive peripheral processors that relay on a host computer, such as a DEC VAX, for input and output as well as hosting all the program development facilities for the array processor. Since array processors are extremely cost effective for floating-point intensive calculations, they are often used in applications that might otherwise be too consuming or costly to carry out on the more general purpose computer hardware such as the VAX or Cray. In this paper we will focus on three main topics. The first is what an array processor is and how achieving its high computational speed has resulted in a complex hardware architecture. The second is our company's experience in interfacing array processors, normally considered real-time devices, under UNIX and how we have been able to convert and support their extensive software development packages for different host computers under various versions of UNIX. And lastly, we will examine the motivating factors for using array processors and the advantages and inherent difficulties associated with their use.

What is an Array Processor?

There are three ways to build faster computer hardware. The first is to increase the speed of the logical elements themselves; the second is to horizontally spread out the calculation among many parallel elements all working at once; and the third is to vertically spread out the steps of the calculation in time along a pipeline, so that many successive different calculations can trickle down the pipeline together. Very recently the emergence of VLSI designed special purpose hardware and the low cost of powerful microprocessor chips has resulted in products trying to incorporate many of these techniques to provide more cost effective and powerful computers. However, these techniques are not new and have been around for many years in the guise of array processors.

Most available array processors are special purpose computers which combine all three approaches, logic speed, parallelism, and pipeline, but, in most cases, pushes none of these to its extreme limit. It is designed for very fast processing of floating-point arithmetic. In comparison, logical instructions are much slower and more cumbersome. They are not an array of processors

UNIX is a trademark of Bell Laboratories.

as many infer from the name, but a single parallel and pipelined processor whose architecture is easily exploited for array-type operations. They typically deliver from 10 to 100 million floating-point operations per second (megaflops). This speed puts array processors in the same computational class as many larger main-frame computers and, in some cases, even supercomputers.

While array processors provide the same general capabilities, the implementation, speed, programming language, and cost vary among array processor manufacturers. Table I contains a summary of the array processors that are currently available by the various manufacturers.

Table I. Currently Available Array Processors.

Manufacturer	*Model*	*Speed in Megaflops*
Floating Point Systems	FPS-5000	8-62
	FPS 164/564	10-300
Numerix	Mars 432	30
CSPI	MiniMap	7
	MAP-400	24
	MAP-6400	5
Star Technologies	ST-100	100
Sky Computers	Warrior	15
Analogic	AP-500	10
Computer Design & Applications	MSP-3000	6

The actual design of each of the array processors differ drastically. Some provide synchronous operation, where the parallel processing units operate independently but use a common clock and cycle time. Other are asynchronous where each individual processing unit operates independently and communicates with the other processing units with slower interfacing logic. A few array processors hide the pipeline nature of the hardware by not allowing explicit control in its assembly language over each of the stages in the arithmetic unit, thus making the machine appear as a more traditional computer at the expense of maximum efficiency of the microcode. In most of the cases where a range of speeds is shown in Table I, the increased speed is gained by the addition of one or more co-processors. These co-processors are essentially additional independent single board array processors that interface to the primary array processor over its data busses. Most array processors have their own private high speed data and program memories (which are generally separate and distinct) and do not use or share memory with the host computer.

While there is much debate as to the advantages and disadvantages of the various implementations, in general, the manner and level of effort one is willing to expend in programming the array processor has a greater outcome on the overall performance one will achieve than the actual architecture or design of the array processor. For cases that do not naturally split into separable computational tasks, taking full advantage of co-processing array processors can be difficult and one may find it more advantageous to have all the computational power in one main array processor unit. Although for separable problems, the co-processors with their separate memories and data paths can provide significant advantages.

The complexity of an array processor can best be seen by examining its assembly language. Shown below are few lines of code for the FPS-5000 array processor:

CYCLE 1: add 2,3; setma; inctma; fadd dpx(0),dpy(1); fmul tm,md; dpx(2)⟨md
CYCLE 2: fadd; fmul
CYCLE 3: dpx⟨tm; dpy(1)⟨fa; fmul
CYCLE 4: dpy⟨md; dpx(1)⟨fm

The first line of code is executed in a single cycle of the array processor and shows the parallel nature of the architecture. The *add* opcode causes an integer add of the contents of the integer (address) registers 2 and 3. The result of the add is used as an address with which to start a main data memory fetch by the *setma* opcode. The *inctma* opcode causes a fetch from table

memory (really another separate data memory) to be initiated at the next sequential address relative to the previous fetch from that memory. The *fadd* opcode causes the floating-point add of the contents of the two floating-point registers *dpx(0)* and *dpy(1)* to be initiated. Likewise, the *fmul* opcode causes the floating-point multiply of the result of a previous fetch from table memory, *tm*, and main data memory, *md*. And lastly, the result of the previous fetch from main data memory is stored into the floating-point register *dpx(2)*. This single line of code causes each operation to be initiated in a separate processing unit of the array processor in the same single instruction cycle. The instruction cycle time is generally on the order of 100 to 300 nanoseconds. To show the pipelined nature of the array processor we have left out all but the pipeline relevant pieces of code from the remaining cycles. In reality, all of the processes shown in the first cycle could be repeated again in any combination in the following cycles. In cycle 2 another floating-point add and floating-point multiply are initiated. These instructions are necessary as the floating-point operations started in cycle 1 must be pushed through the pipeline stages by successive floating-point adds and multiplies until the result is available in the last stage of the pipeline. It is important to keep in mind that although no arguments are shown for the floating add and multiply operations, new adds and multiplies with different operands could be initiated in each of the following cycles. In cycle 3 the result of the floating-point add, *fa*, initiated in cycle 1 is available from the two stage floating-point adder and is stored in a floating-point register. As can be also seen in cycle 3, fetches from memory are also pipelined. The data from the table memory location, *tm*, which was requested in cycle 1 is first available from the two stage table memory unit in cycle 3. In cycle 4 the results of the three stage main data memory unit, *md*, and floating-point multiplier, *fm*, which were initiated in cycle 1 are available and stored in floating-point registers.

Hopefully, several things become apparent from the above example. The first of which is that due to the complexity of the array processor generating optimumized and efficient code for an array processor is a very difficult task. There are a few compilers available for array processors but they are all for Fortran source code input and the better ones are only available for the most expensive array processors. As might be guessed, since arithmetic operations generally span over multiple instruction cycles, branching based on the outcome of these arithmetic operations is very slow and cumbersome. This generally means that the compiler does best when the input code has calculation loops without logic or branches that cause the compiler to have to generate code to flush the pipelines or unfold any tight loops it was able to generate. All of the currently available compilers tend to have drastically different efficiencies dependent on the particular algorithm the input code implements and the amount of hand tuning of the Fortran code that was done based on a knowledge of the quirks of the array processor compiler. This is not to say that logic intensive code cannot be implemented with great efficiency on an array processor, but it is likely that such code will have to be written directly in the array processor's assembly language to achieve the desired level of performance. Although this task may initially seem forbidding, due to the limited number of opcodes and restrictions imposed by the hardware, it is not as bad as one might think.

Another point that is apparent is that the interconnect or busses between each of the functional units is of extreme importance. When the first single chip pipelined floating-point arithmetic units were announced, many "old-timers" in the array processing industry thought that the future of array processors was limited as now almost anyone could design array or special purpose floating-point processors. This, however, has not become the case. Instead, it has been found that the design and implementation of the arithmetic portions of the array processor was not as critical as the interconnect between the various functional units. To make full use of the potential provided by the single chip arithmetic units enough independent data paths for operands and results must be provided to keep the chip processing data at its full speed. It is the complexity of building a network of busses and data paths to meet these needs that array processor manufactures have a great deal of expertise and have been able to use these chips to their full capabilities where others have not.

Because it is a peripheral processor, the array processor requires a host computer to initiate data and program transfers to and from it. Thus the nature of the host interface and host operating system are extremely important factors. Due to the high memory bandwidth required to keep the arithmetic portions operating at full speed, most array processors have their own private data and program memories. Although some manufacturers have tried shared memory approaches to avoid the overhead associated with having to transfer all data and program code to and from the array processor and the host computer, the drawbacks in terms of bandwidth, bus contention, and operating system implementation (especially in a virtual memory environment) generally have outweight the advantages. A special case might arise for the lower speed array processors where memory bandwidth is not as critical especially when the array processor is designed to interface only to a specific host computer running a specific operating system.

Array Processing Under UNIX

There are obstacles, however, that must be overcome in the use of array processors under UNIX. The first of which is finding an array processor that is supported under UNIX, or, lacking that, any software that will allow the array processor to function in the UNIX environment. To our knowledge, there is currently UNIX software and support available for only one of the array processors shown in Table I. The reasons for this are complex. The foremost is that most array processor manufacturers do not see the UNIX community having a high potential for sales of array processors. This is due to the tradition that most scientific number crunching has been done almost exclusively in Fortran and mostly not on UNIX based systems. Due to their own limited software engineering resources, array processor manufacturers do not feel they can justify the effort to convert their software to what they see to be a low potential market and, in the process, have to make substantial changes to provide a *C* language interface as well as deal with all the problems of UNIX *f77*.

This type of attitude by hardware manufactures has been experienced in the past by the UNIX community (for example, by disk controller manufactures, among others) and, for the most part, the UNIX community has responding by buying the product without any software support and writing all the necessary device drivers themselves. If the array processor required just a device driver, a similar situation might occur and one would see more array processors running on UNIX host computers. However, a typical array processor comes with an extensive program development software package that usually exceeds 150000 lines of Fortran code. This Fortran code, written by the array processor manufacturers, tends to be rather non portable to the UNIX *f77* environment. Thus the purchase of an array processor without software is to undertake a substantial (multi man-year, in most cases) software development or conversion project as well.

Apunix Computer Services has been supporting a line of Floating Point Systems array processors with UNIX software for over four years. This task has been particularly challenging for two reasons. The first of which is the design and implementation of an efficient device driver and user interface for the array processor. The second is the task of supporting a large volume of software, including a device driver, under almost all currently existing versions of UNIX and different host computer manufacturers.

UNIX, being a multiuser multiprogramming system, has been criticized by array processor manufacturers for its overhead in handling real time applications such as encountered in trying to maintain a sufficient flow of data in and out of an array processor. If care is not taken, the speed of the array processor can be lost in the overhead of servicing it. Over the years, several approaches to the UNIX device driver have been taken. Initially, our driver used two minor character devices for array processor. One minor device was a DMA (direct memory access) path into the array processor's main data memory. The other was used for the loading of the various interface registers of the array processor in a block fashion. Even in this version of the driver a great deal of care was taken to avoid the expense of a system call each time a device register needed to be accessed. Most array processor interfaces require multiple accesses to many of the

interface registers to load and initiate any program execution in the array processor. This particular implementation had two major flaws. The first being that since *read* and *write* system calls were used to communicate with the device (to simplify the DMA portion of the driver by automatically insuring "physio" would be called) it was also possible to easily crash the system by uncontrolled access to the device nodes. In addition, since the DMA and register access were considered separate entities it was impossible to buffer commands and avoid system calls when any data transfer to or from the array processor was involved. The current version of the driver solves both of these problems by using only one minor character device entry for each hardware device. All communication between the user program and the device is accomplished by passing command blocks which can contain multiple commands of any type (both DMA and register access) with one system call. The command block is passed via an *ioctl* system call to avoid any problems with any unintentional user access of the device entries. In addition, we have found it necessary to add "in-driver" queuing of processes wishing to use the array processor to avoid substantial dead time between processes which use the array processor for only very short amounts of time. This has provided a level of efficiency in the use of the device which we feel is comparable, and in some cases superior, to that achievable under VMS.

Portability of the software to not only virtually all available versions of UNIX but also to different host computers has also been a challenge. We have, however, been able to support these different combinations with only one version of the source code. We are able to supply a single device driver that contains all the appropriate *ifdef*'s to run under all common versions of UNIX (Berkeley, Bell, Version 6, and Version 7) and three different computer architectures (PDP, VAX, and Perkin Elmer). We have found two key steps in writing a portable device driver. The first is to make extensive use of the resources available in the UNIX kernel *bio* support software (*viz.*, the "physio" routine for validating and setting up DMA's) and the second is to initially write the driver for a BSD version of UNIX. The BSD version of the driver will have one structure, *uba_device*, that must be carried through to a limited extent in the other versions of UNIX. This can be accomplished by defining a small portion of this structure for the other versions of UNIX as shown below:

```
struct uba_device {
        char            *ui_addr;       /* address of device in i/o space */
        short           ui_alive;       /* device exists */
};
```

Using *ifdef*'s, the address of the device and its existence can be filled in when the device open routine is entered from either a set of defines (Version 6 and 7 UNIX) or a global configuration array (Bell System III and V). The addition of these two fields makes the device register access syntax the same for all versions of UNIX. All of the rest of the minor differences such as slightly different names in the buffer header structure, using "ubmalloc" and "ubmfree" instead of "ubasetup" and "ubarelse", and the number of parameters to "physio" can be easily handled by a few simple *ifdef*'s in the driver code.

The device driver is, however, the smallest portion of our software package. Most array processors come with a host of software that includes assemblers, linkers, simulators, debuggers, and executive routines. Most of the array processor manufacturers write this software exclusively in Fortran and gear the user interface to be similar to more archaic software environments such as VMS. Given the poor portability of Fortran source code (particularly to UNIX systems) we have found it necessary to completely rewrite (and in many cases redesign) the software in *C*, *lex*, and *yacc*. This has not only allowed us to provide a user interface consistent with the UNIX environment, but also incorporate into the software many of the powerful capabilities found in UNIX, such as having all array processor source code pass through the C preprocessor thus adding "include file" and "define" capabilities. It also allows us to provide our users with a maintainable set of source code that is easily enhanced or fixed by someone with a good knowledge of C.

Why Use an Array Processor?

The major reason that the use of array processors is attractive is its cost effectiveness. This is shown in Table II form the work of Bucy and Seene[1] and is described in detail by Karplus and Cohen.[2] Columns 1 and 2 show computers that have been commonly used in floating-point intensive applications and their theoretical speed. Columns 3 and 4 show the performance in terms of time and achieved megaflops for a sample application. Columns 5 and 6 show the motivating economic reason for using array processors. The array processor, as represented by the FPS-5000 in this table, has the lowest cost per achieved megaflop and one of the lowest overall installation costs.

Table II. Comparison of Processor Cost/Performance for Demodulation Problem.

Machine	*Maximum theoretical megaflops*	*Time per iteration (ms)*	*Achieved megaflops*	*Approximate cost (dollars per flop)*	*Installation costs (millions of dollars)*	*Software development time (man-months)*
Cray-1	80-140	2.1	38.4	0.21	8	0.5
Star-100	25-50	4.9	16.8	0.48	8	2.0
Illiac IV	40-80	9.0	9.1	1.10	10	3.0
FPS-5000	6-12	13.9	5.9	0.03	0.15	6.0
CDC 7600	5-15	25.0	3.3	0.91	3	1.1
IBM 370-168	2-4	94.0	0.87	2.30	2	1.0
CDC 6600	1-3	130.0	0.63	1.59	1	1.0
VAX-11/780	0.5	311.0	0.26	0.77	0.2	0.5
PDP-11/70	0.2	870.0	0.09	1.67	0.15	1.2

However, the use of array processors also carries with it a substantially higher program development time, in addition to a costly learning curve if users must become familiar with its unusual and challenging parallel assembly language, as shown by the last column in Table II.

Although there are Fortran compilers available for some array processors, except in a very few cases the compiled code either tends to be too large and linear to be efficient and held in a sometimes limited amount of program source memory (which is different than data memory for most array processors) or the optimization is highly dependent on the structure of the Fortran source. The best compilers are, of course, most readily found for the most expensive array processors. Although the optimum use of the array processor may involve programming in its assembly language, most applications utilize the array processor by making calls either from a *C* or *f77* program running on the host to an extensive set of library routines, which have been written in the array processor's assembly language by the array processor manufacturer. These libraries often consist of hundreds of routines from simple vector and matrix functions such as a "vector add" to very specific and special purpose forms of convolution, filtering, and integration algorithms. In addition, a *C*-like high level language is available to write outer loops linking calls of the individual library routines together in a form that is compiled and linked to be executed as one call to the array processor. Using this technique, when a good enough match between the combination of library routines and the solution of the problem can be found, most users do realize a substantial improvement in the execution time using the array processor over that of their host computer alone and are also able to offload a sizable computational burden from their host system.

Even with the advent of the newer "cheaper" vector processors, we believe that for many years to come array processors will still offer a substantially more cost effective solution to many floating-point intensive calculations. An example of the computational power an array processor offers can be seen in the field of computer simulation of chemical systems. In the computer aided design of drugs, a chemical system consisting of hundreds of atoms must be simulated by calculating the forces each atom exerts on each other atom and numerically integrating the equations of motion of the system over thousands of time intervals to see how a particular molecule will interact with other molecules.[3]

In Table III, column 1 shows the various computers for which some benchmarks for this application of array processors were performed. Columns 2 and 3 compare the speed of the array

Table III. Comparison of Molecular Dynamics Simulation Times on Various Computers

Computer	*In C, w/o FPA on VAX UNIX (BSD)*	*In Fortran, w/ FPA on VAX VMS*
FPS-5000	1	1
VAX 11/780	80	35
VAX 11/750	150	
PDP 11/40	300	

processor (a FPS-5000 in this case) under two situations. Column 2 is a direct translation of the array processor code into *C* and run on VAXes without floating-point accelerators (FPAs) under UNIX. Column 3 is an equivalent optimized Fortran version of the code running on a VAX with a floating-point accelerator under VMS. As can be seen from the table, the array processor implementation is approximately 35 times faster than a VAX 11/780 with a floating-point accelerator and an optimized Fortran compiler. Thus a simulation that can be run in a week and a half on the array processor would take a year on a VAX even if the VAX were totally dedicated to that calculation. Although a vector computer such as the Cray might perform the calculation in similar or less real time, full time usage of an array processor is equivalent to 2-8 hours per day of Cray-1 time at a fraction of the cost. Thus array processors do indeed make feasible calculations which might otherwise be too time consuming or costly using other forms of computer hardware.

References

1. R. S. Bucy and K. D. Seene, *Comp. Math. Applications*, Vol. 6, p. 317 (1980).
2. W. J. Karplus and D. Cohen, *Computer*, p. 11 (September 1981).
3. P. H. Berens and K. R. Wilson, "Molecular Mechanics with an Array Processor," *Journal of Computational Chemistry*, Vol. 4, No. 3, pp. 313-332 (1983).

Integral Array Processing in a Multiprocessor UNIX Environment

Denise Hewson
Gregory Cullen
Alan Nugent

MASSCOMP†
Software Engineering
One Technology Park
Westford, MA 01886

ABSTRACT

This paper explores an integrated approach to computationally intensive problems which have traditionally involved the use of a peripheral array processor. This approachcombines the performance gains of a multiple processor system with the inherent advantages of a single UNIX operating system to manage all resources.

Two vector/array intensive applications will be presented as case studies which exemplify the potential range of problems which lend themselves to the single system approach.

Introduction

The integrated system under discussion here is the MASSCOMP MC-500. We will first give you a brief overview of the system's architecture, then a more detailed description of the MASSCOMP AP-501 Array Processor's hardware and software. Finallly, we will look at how this system is being utilized by two of MASSCOMP's customers, NMR Imaging and Princeton University.

System Architecture Overview

MASSCOMP four principal markets: realtime data aquisition, scientific computation, CAD/CAM/CAE, and imaging. Since most applications in these markets are computationally intensive, MASSCOMP introduced the AP-501 integral array processor in 1984 as an option on its 500-Series computers and engineering workstations.

An MC-500 system (Figure 1) consists of a 32-bit VLSI virtual memory computer with up to 6 megabytes of ECC memory, a 4 kilobyte cache, a 5 1/4" floppy disk, Winchester disk drives of from 50 to 474 megabytes, and 1/4" or 1/2" tape drives.

---

† MASSCOMP and RTU are Trademarks of Massachusetts Computer Corporation.
UNIX is a Trademark of AT&T Bell Laboratories.
Multibus is a Trademark of Intel Corporation.

The system has a triple bus architecture. An enhanced performance Multibus is used as a general- purpose bus for peripherals such as disks, tapes, ethernet, and up to 4 independent graphics processors (IGP). Proprietary modules such as the CPU, ECC memory cards, hardware floating point processor (FPP), and array processor (AP) are coupled on the high speed (8 megabytes/second) proprietary memory interconnect bus (MI). The third bus in the system is the STD+ bus. It is a enhanced version of the STD bus and allows communication between the data acquisition control processor (DA/CP) and any standard STD modules.

Dual processor capabilities are achieved on this system by adding an additional CPU card to the proprietary memory interconnect bus.

The MC-500's operating system, RTU (Real-Time UNIX), was originally a derivative of Bell System III. The system was then enhanced with 4.2 BSD style virtual memory management and copious utilities from 4.2 BSD as well. For example, a full job control C-Shell is supported.

Since the mainstream of MASSCOMP's business is in the scientific and engineering communities our UNIX was modified to provide special real-time capabilities. Those modifications included:

- Changes to the scheduler to allow for absolute, unchanging priorities.
- Modification of the file system to allow for contiguous disk files.
- Provisions for locking down (making non-pageable) all or part of a process.
- Addition of AST (Asynchronous System Traps), the software analog of a priority hardware interrupt.
- Alterations to device drivers to allow transfers directly to user memory (without going through kernal buffer space).

With the 2.2 Release of RTU the kernel is now a derivative of AT&T System V Release 2 complete with semaphores, message queues and dynamic shared memory regions. The operating system has also been embellished with University of California's 4.2 BSD style Local Area Networking. A great number of utilities from both UNIX camps are still provided. The 2.2 Release of MASSCOMP RTU also brought support for multiple CPU's in a single MC-500 chasis.

Array Processor Architecture Overview

Differences Between Integrated and Peripheral APs

The following is a list of how our integrated AP differs from a peripheral array processor:

[1] Because only one vendor is involved, no effort is required to integrate the array processor into the system. There is no hardware interface, and the driver for the AP is provided with the system. The user can just plug in the board, install the AP software library and start writing code.

[2] There is no need for separate AP programs for I/O processing and data processing. Only one program is required, and this program resides in the host. The program can contain calls to the AP library mixed in with other program statements.

[3] AP programs are written in a high level language (C, Fortran or Pascal-2). The user does not have to microcode his application.

[4] Additional software in the form of assemblers, linkers, loaders, debuggers, and executives are not required. The system's assember, compilers, loader, and debugger are utilized.

Considerations for Using the AP-501

The following is a list of things to be taken into account when using the AP-501.

[1] The AP-501 can only be used by one process at a time.

[2] In coding a program, the user must consider the following:

- How best to take advantage of the ability to overlap DMA and arithmetic processing in the AP.
- How to manage storage of his vector data in the AP. For more complicated functions like large vector FFT's this is taken care of for him. Vector data remains in the AP until the user retreives it (it is not returned to the host after each AP instruction).

[3] The system's debugger only allows the user to look at the AP's vector memory, ring buffer, and communication block. For this architecture this is sufficient.

[4] Because of the limited size of the AP-501's instruction ring buffer, it is not possible to do chaining (loading multiple AP instructions with one subroutine call) on our AP.

[5] Currently there is no simulator for the array processor.

AP-501 Hardware Organization

The AP-501 is a closely coupled single board processor on the 500-Series memory interconnect bus, and communicates with other processors via a multiprocessor UNIX operating system.

As shown in Figure 1, the AP-501 array processor attaches directly to the 500-Series bus connecting the CPU and memory. With the AP-501 on the bus, the CPU can access the array processor as rapidly and directly as main memory. For most applications, peak data transfer rates exceed 5 megabytes per second. This architecture allows concurrent processing in the system CPU and in the AP-501 to be overlapped with data transfers to array processor vector memory.

The AP-501 uses a 10Mhz control processor (Figure 2) to coordinate its operations. It initiates DMA data transfer activity and the concurrent execution of arithmetic operations. It also initiates virtual to physical address translation, and initiates system interrupts.

Instruction packets sent from system memory to the AP-501's control processor are first loaded into a ring buffer. The control processor unloads as instruction packet from the ring buffer, reads it and converts it into array processor instructions. The control processor also coordinates the 5-stage pipelining of instructions within the math engine. The overlap of DMA transfers from system memory and computation in the math engine is also managed by the control processor.

On board vector memory consists of 16,352 32-bit registers. Vector memory appears to the CPU as an extension of its own address space, allowing the direct manipulation of vector memory with CPU instructions. Close coupling to the CPU's virtual memory environment aids in manipulating arrays larger than physical memory. Partitioning of virtual memory and page translations are transparent to the user.

The AP-501 math engine performs single precision floating point functions on data in vector memory. It reaches speeds of 10 million floating point operations per second. Its Adder and Multiplier operate concurrently and are pipelined in five stages. Each of the five stages is measured at 200 nanoseconds. The math engine also converts integer to float values when required.

AP-501 Software

Programs running on the main CPU under UNIX access the AP-501 by a set of subroutine libraries. Calls to the subroutines can be issued from C, FORTRAN, and Pascal-2.

The subroutine libraries make calls to four classes of instructions which comprise the instruction set:

Direct Memory Access: to move all data types between system memory and the AP-501.

State-Change: to change rounding modes, default interrupt handling and to read current defaults.

Arithmetic: to convert data from float to integer (and reverse), to move data within the AP, basic arithmetic operations, complex multiply, FFT, vector comparisons, dot products, scatter and gather, MIN/MAX and others.

Synchronization: to coordinate the operation of the AP, the math engine, and the DMA unit with the AP, and the operation of the main CPU wiht the AP.

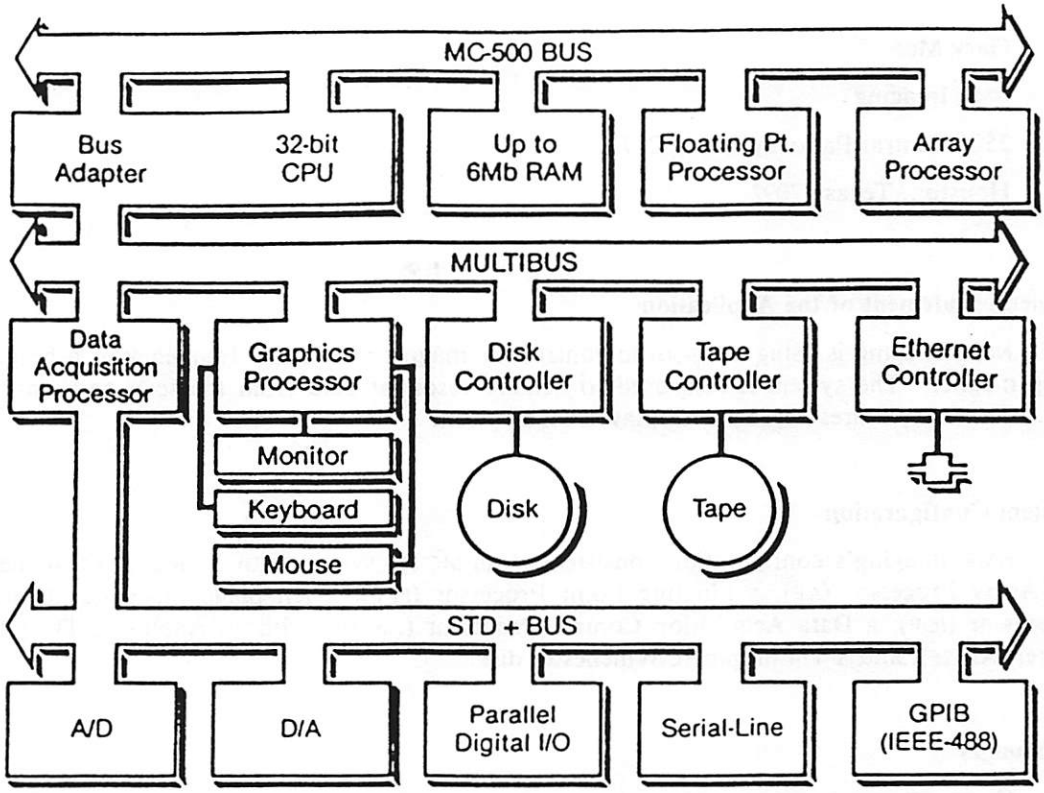


FIG. 1

500 SERIES SYSTEM ARCHITECTURE

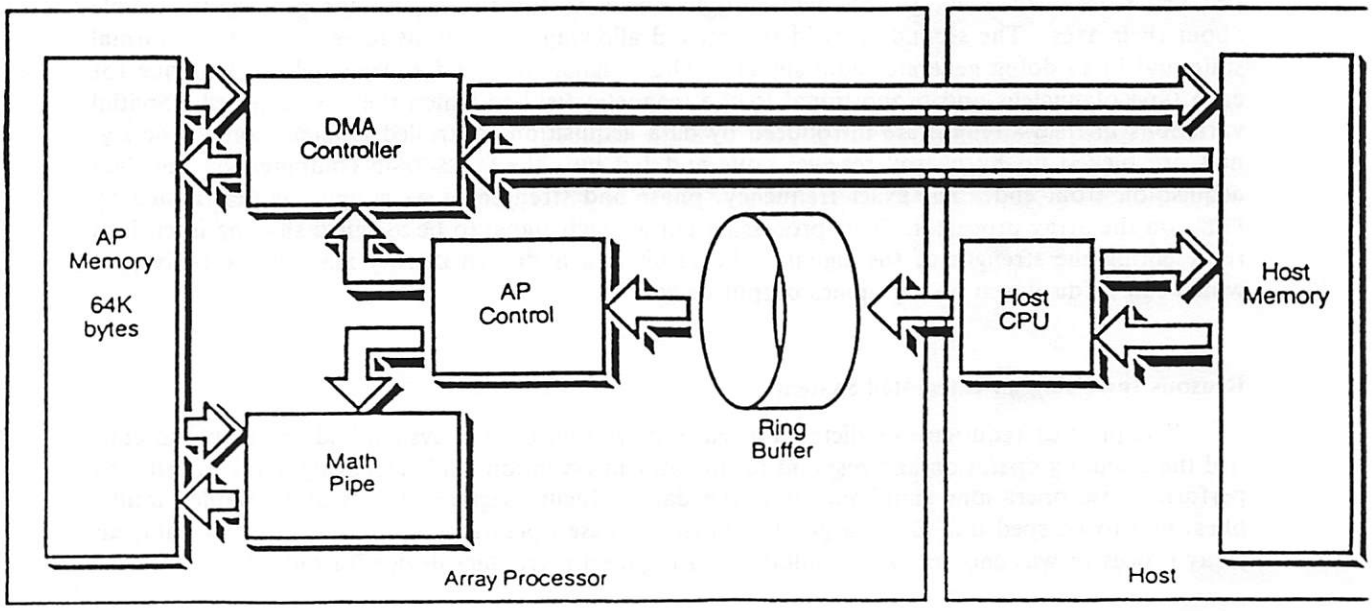


FIG. 2

PROGRAMMER'S VIEW OF THE ARRAY PROCESSOR

OEM Application

Gary Mee

NMR Imaging

2501 Central Parkway, Suite C17

Houston, Texas 77092

General Statement of the Application

NMR Imaging is using a MC-500 to control the magnetic field and frequency of a brain scanning magnet. The system is also used to acquire resonant data from the scan and reduce that data to create pictures which are displayed on a screen.

System Configuration

NMR Imaging's configuration consisted of an MC-500 system with 3 megabytes of memory, an Array Processor (AP), a Floating Point Processor (FPP), a ten plane Independent Graphics Processor (IGP), a Data Acquisition Control Processor (DA/CP) with an Analog to Digital Converter (AD12F), and a 474 megabyte Winchester disk.

Languages

C

Actual Processing Involved

NMR Imaging does its magic by putting two magnetic fields near the part of the patient to be examined. The field in one direction aligns the protons in the nuclei of the atoms in the subject being observed, while the RF field at right angles to the first causes the protons to wobble about their axes. The second RF field is removed allowing the protons to return to their normal state and in so doing generate radio signals. These signals have a frequency which is unique for each type of nucleus and proportional to the magnetic field to which they are exposed. Spatial variations in field strength are introduced by data acquisition controlled gradient coils. The signals are picked up by nearby receiver coils and fed into the MASSCOMP computer via the data acquisition front end. The exact frequency, phase and strength of each signal is determined by FFT's on the array processor. This processing allows each signal to be assigned shading intensities representing the strength of the signal. The result is a hydrogen density map of the body part which can be displayed on a graphics output device.

Reasons for Using an Integrated System

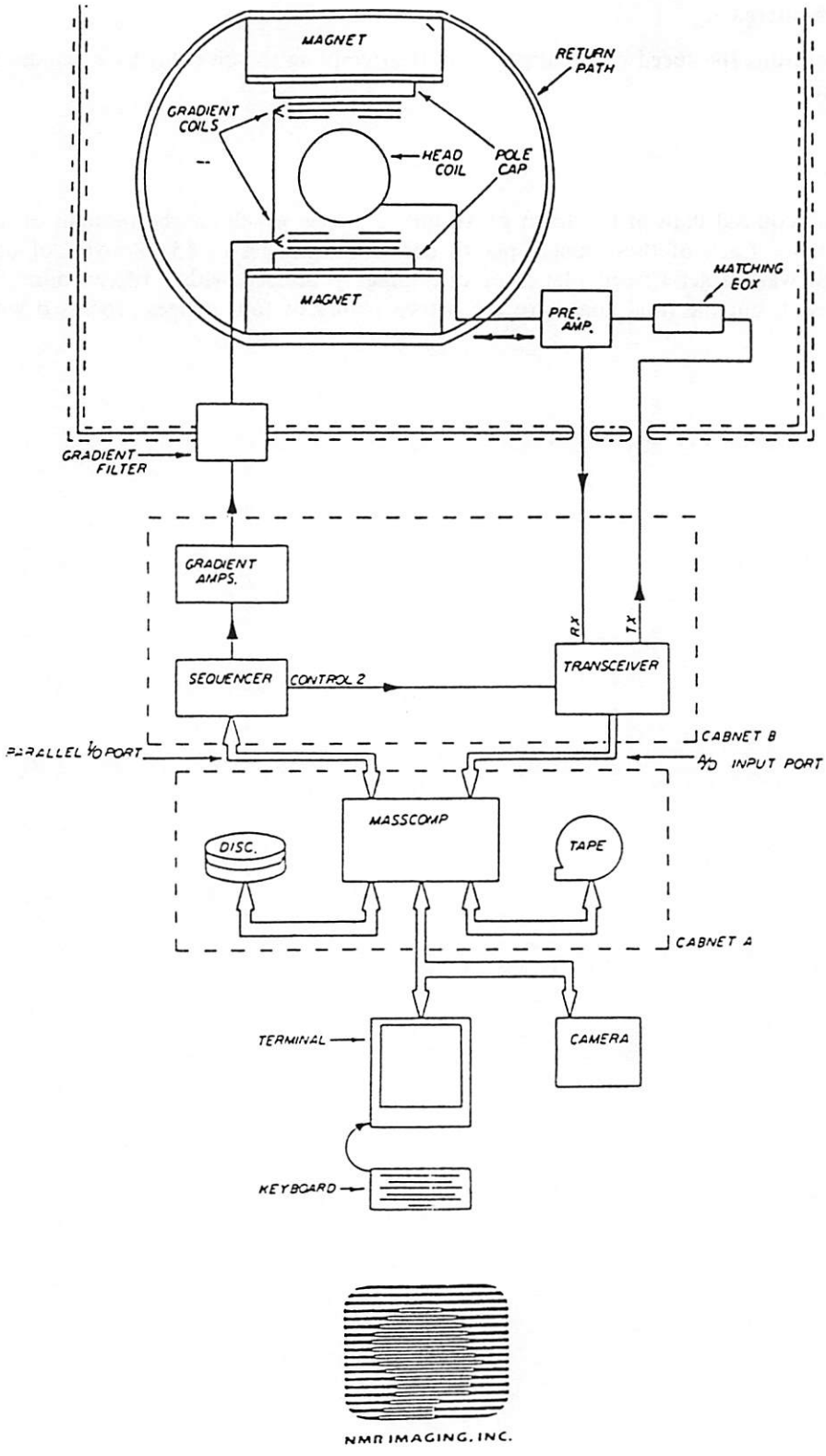
The product requirements dictated a real time response. The system had to be able to control the scanning operation and respond to operator intervention while acquiring data. In order to perform these operations simultaneously, the data reducing segment (FFT's and complex multiplies) had to be sped up. Since a good number of these operations were on vectors of data, an array processor was chosen. The resultant data required a graphics device for output.

Problems Encountered

The AP outruns the speed of the disk. NMR is attempling to solve this by expanding the use of virtual memory.

Goals/Results

The scan acquired data in the form of 16 image planes which can be thought of as a slice through the skull. Each of these image planes contains 8 images or 65 kilobytes of data. An acceptable goal was to get fifteen planes of one image processed within fifteen minutes. This goal was achieved, but the final goal is to get fifteen planes of four images processed within this time frame.



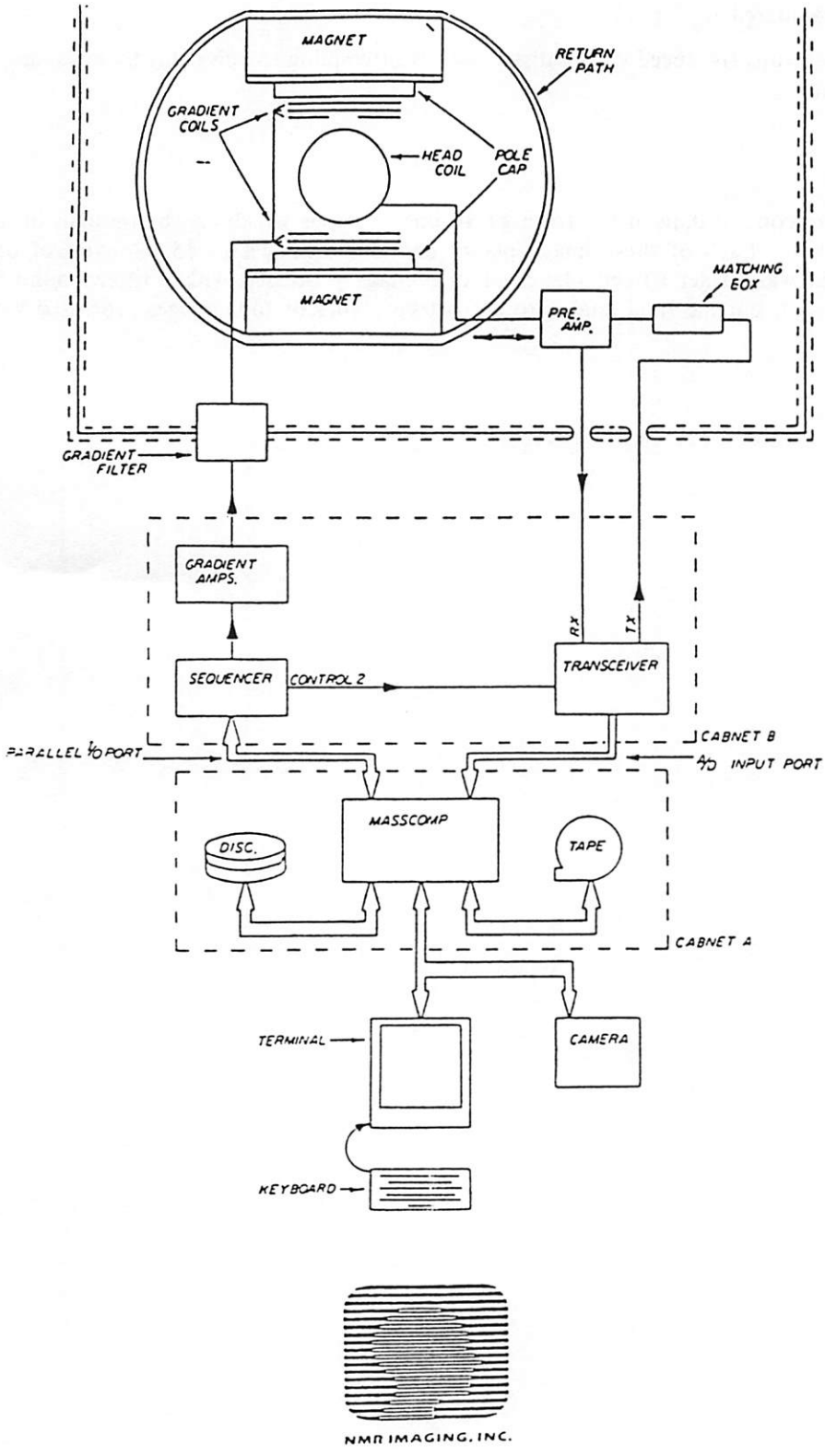
NMR IMAGING, INC.

Problems Encountered

The AP outruns the speed of the disk. NMR is attempling to solve this by expanding the use of virtual memory.

Goals/Results

The scan acquired data in the form of 16 image planes which can be thought of as a slice through the skull. Each of these image planes contains 8 images or 65 kilobytes of data. An acceptable goal was to get fifteen planes of one image processed within fifteen minutes. This goal was achieved, but the final goal is to get fifteen planes of four images processed within this time frame.



NMR IMAGING, INC.

University Application

Grant Stokes

Department of Physics

Princeton University

Princeton, New Jersey 08544

General Statement of the Application

Grant Stokes, a PhD candidate in the Department of Physics at Princeton University required a system to process and reduce radio telescope data for his research on pulsars.

System Configuration

Princeton's configuration consisted of an MC-500 with 4 megabytes of memory, an Array Processor (AP), a Floating Point Processor (FPP), 2 Cipher Tape Drives, 2 Visual-55 Terminals, and a Citoh printer.

Languages

FORTRAN, C, and 68000 assembler

Actual Processing Involved

Just a little background: pulsars are rotating neutron stars that have a strong off axis magnetic field. Narrow cones of high power radio waves are emitted from the magnetic poles and swept lighthouse fashion past the earth. Therefore, we observe a radio signal pulsed at the star's rotation period (between 1.5 milliseconds and 4 seconds). In addition, since the propagation velocity in the galactic medium of the signals is a function of radio frequency, the pulsar signals are dispersed. In the past, generic pulsar processing involved taking a time series of data and "de-dispersing" it with certain arbitrary time delays between frequency channels. Each time delay represented a dispersion step. Because of compute costs, only a limited number of dispersion steps were taken, usually no more than eight. This processing did not take full advantage of the data.

Grant's analysis closely parallels generic pulsar processing with one important exception. Instead of inserting arbitrary delays between frequency channels to compensate for dispersion, a two dimensional Fourier transform was used to search for the best period-dispersion combination.

Radio wave data were gathered from NRAO's 300 foot telescope in Green Bank, West Virginia. 32 channels of data were digitized at a rate of 500 samples per second per channel. The data were then packed and stored as 32 x 64K arrays (64K time samples for each of the 32 channels) on magnetic tapes for later analysis at Princeton.

Each array of raw time samples was unpacked from tape and Fourier transformed along both axis to produce a complex array with axes of phase and frequency.

Since a given dispersion delay, delta-T, produces a progressively larger phase shift in higher harmonics, power at a given dispersion will lie on a line of constant slope. The larger the dispersion the larger the slope. Dispersions with slopes less than the diagonal were covered with full period sensitivity. Larger dispersions were covered with reduced sensitivity. Herein lies the

advantage of the two dimensional Fourier transform analysis method. No information was thrown away by arbitrarily quantizing the dispersion compensation before the FFT's. Dispersion coverage was immensely enhanced on the high dispersion end where the data would be lost in a generic search using direct dispersion trials.

Once the array was Fourier transformed along each axis the result was saved on disk and a magnitude array was formed. The magnitude array was flattened to remove the low frequency (1/f) rise. Persistent periodic interference was removed from the power spectrum at this point. The computer then examined all the harmonically related points that lay on lines of constant slope. The number of harmonics inspected was either 1, 2, 4, 8, or 16 depending on how many were available in the power spectrum for the period examined. The best few hundred period-slope combinations with the greatest power above the noise were saved for later consideration.

Once a list of periods that exhibited power above the noise was compiled, each entry was checked for signal strength. For each period-DM entry in the list, all the harmonics were retrieved from the complex array stored on disk. These harmonics were inserted into an array and inverse transformed to produce a signal profile. If the power in the harmonics was from a periodic pulse the profile would contain a sharply peaked pulse. The signal-to-noise ratio of all the profiles generated by examining the list of suspects was measured and the largest one for the scan was saved. If the best signal-to-noise for a given scan exceeded a threshold (usually 8.0) the profile was printed out as a possible candidate. The printout was inspected by hand and a subjective determination was made to pick the candidates for re-observation.[1]

Reasons for Using an Integrated System

Princeton considered an integrated system for the following reasons:

[1] They only had to deal with a single vendor.

[2] Their application was so computationally intensive that timesharing would be aproblem. The cost of the MASSCOMP system with its array processor was such that they could afford to use it as a dedeicated system.

[3] The performance gain provided by the array processor made the time frame for the project realisitc.

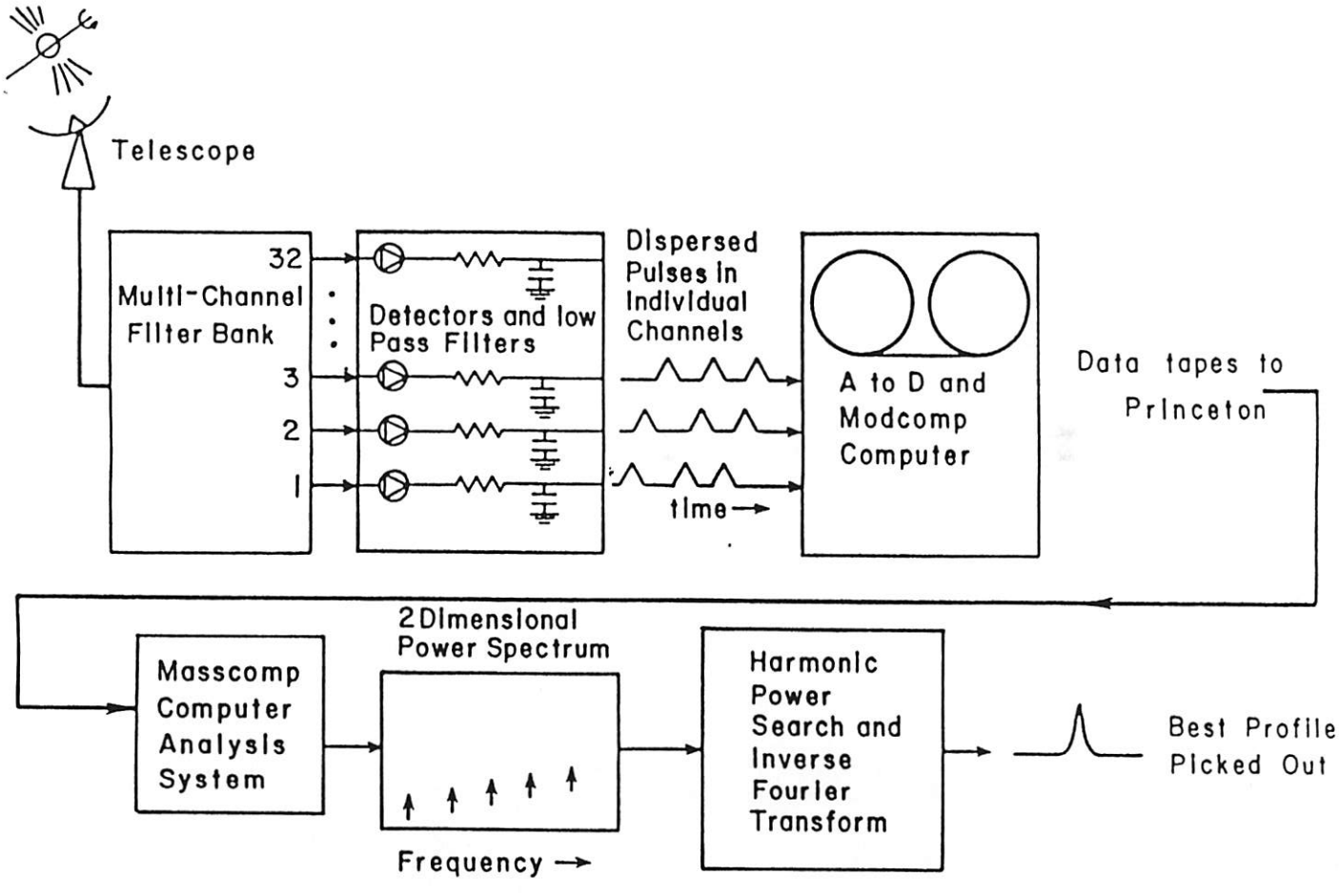
Performance Information

Grant had determined the 360 mag tapes of radio wave data would require 6 CPU years to process on a general MASSCOMP system. The addition of a floating point processor to the system reduced the data processing time to 2 CPU years. With the release of the array porocessor the processing time dropped to 2.5 CPU months.

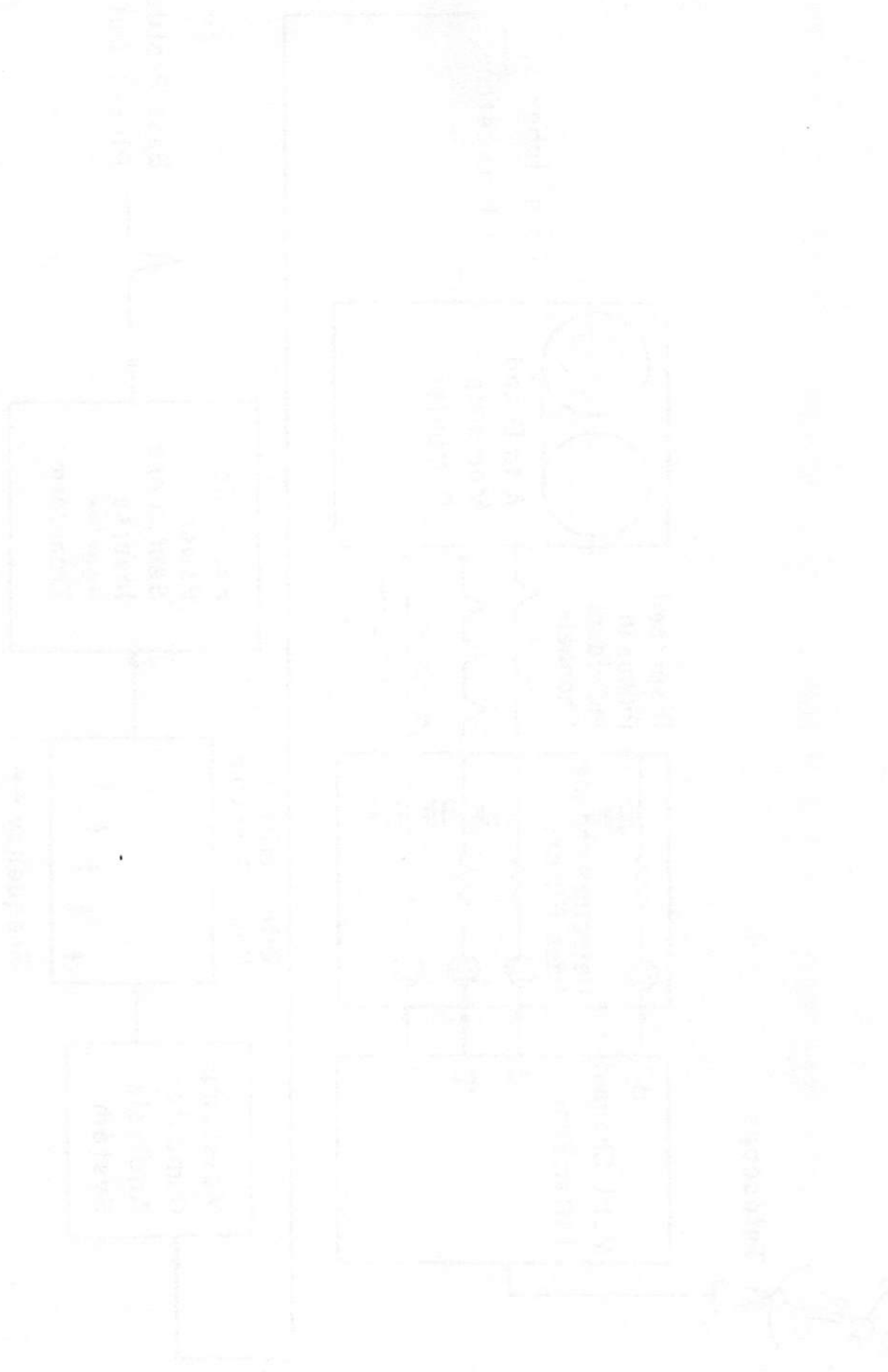
Goals/Results

Since pulsars were first discovered in 1968, approximately 370 have been identified, many under the direction of Dr. Taylor. In the past year with the use of the MASSCOMP system, Grant has been able to identify an additional 20 pulsars.

1. This material was taken from Grant Stokes' PhD thesis on Pulsar Research, Department of Physics, Princeton University.



Princeton Apparatus and Analysis Algorithm



A Parallel Array Processing Environment under 4.2bsd UNIX

Richard Jaenson
Gregory Taylor
Cyrus Umrigar
Alison Brown

The Center for Theory and
Simulation in Science and Engineering
Cornell University
Ithaca, New York 14850

1. Introduction

The possibility of moving computationally intensive jobs off the VAX† is of great interest to those people doing large scale scientific computing. Although the size and complexity of the calculations done by theorists continues to grow, their computing budgets do not necessarily follow suit. Coupled with the often prohibitive costs of Supercomputers and the difficulty of leasing and/or scheduling time on presently available large scale computers, theorists may be forced to either scale down the problems they wish to tackle, or find themselves facing a relatively short period of scheduled Supercomputer time in which a huge task must be accomplished.

The relatively low cost of a VAX750 class machine and the very low cost of 4.2bsd UNIX‡ has combined to make these two products prevalent in today's university computing environment. Unfortunately, the VAX750 is quite slow in terms of number crunching MFLOPS, even when equipped with a Floating Point Accelerator. At the Center for Theory and Simulation in Science and Engineering at Cornell University, we have attempted to move the computationally intensive work off the VAX750 while still providing users with the full networking and editing capabilities that they have have become used to with UNIX. One of the major problems in a theoretical/simulation environment is the relative lack of collaboration among scientists and engineers. Although the specific areas of research may vary widely in nature among the engineering, biological and physical sciences, the computational strategies and techniques for problem solving and simulation are quite similar. To help facilitate discussion among these individuals facing similar challenges, we have established the Theorynet here at Cornell—a computer network based largely on VAX750s running UNIX 4.2bsd and VAX/VMS[1]. One VAX750 resides in each participating department, with a total to date of about 15 nodes. The machines are maintained by the individual departments themselves, with the Theorynet staff providing help in the form of consulting and support of the networking resources available. This dedicated purpose networking environment allows us to address some common concerns and problems within the theoretical (as opposed to experimental) scientific community, and also allows for better communications among theorists of varying disciplines. The community served by the network also benefits by the sharing of any special hardware or software which is attached to any nodes on the network.

---

† VAX is a trademark of Digital Equipment Corporation

‡ UNIX is a trademark of Bell Laboratories

1 VMS is a trademark of Digital Equipment Corporation

2. System Overview

The Parallel processing environment discussed in this paper is one such node on the Theorynet. It is a special VAX750 that runs both 4.2bsd and VAX/VMS (though not at the same time). Attached to a second UBA on the VAX750 is an Aptec Computer Systems DPS2400. The DPS (Dimensional Processing System) is in turn connected to eight Floating Point Systems FPS100 Array Processors. Each FPS100 is equipped with a Unibus[2] interface and could be attached directly to the VAX Unibus, but this would limit the highest I/O throughput per Array Processor to approximately one eighth of the maximum Unibus throughput (assuming that the eight Array Processors were on a dedicated UBA). The true power of the Array Processors would thus be lost due to any heavy I/O overhead. The Aptec DPS2400 provides an alternative to this problem, in that it functions as a very high speed interface between the VAX and any Unibus device. The DPS has a high bandwidth (24 MByte/sec.) Data Interchange Bus (DIB) that can have numerous Unibus devices attached to it. Each Unibus device is connected to the DIB via its own Data Interchange Adaptor (DIA). Up to 24 MBytes of Mass Memory common to all DIAs are also accessable from the DIB (see Illustration 1).

Each of the 10 DIAs in our DPS2400 system is composed of 4 AMD 2901 Bit Slice microprocessors (capable of 6 MIP/sec.) and 4K words of program source memory (48 bits wide). A DIA has both a 3 MByte/sec. private bus interface to its attached Unibus device, and a 12 MByte/sec. DMA interface to the Mass Memory. The DIAs can be software switched between two modes: the *transparent mode* and the *DIB mode.* The transparent mode allows the VAX host to access the Unibus device associated with it in a normal fashion. In the DIB mode, the DIA will intercept all interrupts and DMA activity.

The default mode mode of operation is transparent. This allows standard applications, drivers, and software (such as software from APUNIX, Inc.) to be configured into the system. In the transparent mode, the Aptec DPS2400 system is essentially uninvolved in host—Array Processor transactions, and all throughput is constrained by the 1.2 MByte/sec. speed of the VAX UBA. It is in the DIB mode that a tremendous increase in computational speed can be realized for certain algorithms.

The FPS100 Array Processors are vector machines which can be programmed in assembler or FORTRAN. The communication between an Array Processor and its DIA is accomplished through special subroutine calls from the Array Processor program. After the Array Processor and the DIA have established synchronization, the Array Processor builds a string of work packets for the DIA. Once the DIA has received and decoded a packet, it initiates and controls all data movement transactions into and out of Mass Memory and Array Processor main data memory. The user can write Array Processor applications that utilize all eight Array Processors , but they must partition the program in terms of Array Processor usage themselves, and program in the parallel synchronizations and communications between Array Processors.

The microcode program that normally runs in a DIA with an Array Processor attached to it is called APSLAVE. The APSLAVE program performs the following tasks:

- Reading/writing data from the Array Processor's main memory to/from Mass Memory
- APSLAVE is able to synchronize with other DIAs by waiting on Mass Memory events.
- APSLAVE performs the task of terminating the DIA's connection with the Array Processor and re-establish connections with the host VAX.

---

2 Unibus is a trademark of Digital Equipment Corporation

With a self-contained program running in an Array Processor (once the DIB mode is initiated, the Array Processor is no longer in communication with the VAX host) and the proper microcode running in a DIA, the Array Processor can do DMA read/writes directly into the Aptec's Mass Memory. This is true for all eight Array Processors on our system. Since the total bandwidth of the DIB is 24MBytes/sec. and the eight Array Processors each have a 2 MByte/sec. UNIBUS interface, all eight Array Processors can do simultaneous transfers and still be within the maximum bus bandwidth of the DPS2400. Given an optimal mix of vector arithmetic operations in the Array Processor each FPS100 is rated to run at 8 MFLOPS/sec. This optimal mix would allow both the add/subtract and the multiply pipelines of the Array Processor's internal architecture to operate at maximum efficiency. Assuming an optimal combination, the total computational speed of the DPS2400——FPS100 system could theoretically approach 64 MFLOPS/sec.

The following two examples briefly illustrate how the parallel processing environment can be used for high speed offline calculations:

- **Physics——A Lattice Gauge Calculation:** A 4 MByte 4-dimensional matrix can be loaded directly from the host VAX750 directly into Mass Memory via one large DMA transfer. Each Array Processor can be instructed to do nearest neighbor calculations on one eighth of the total lattice, and all eight Array Processors can be started simultaneously. At the end of the computation, control is returned to the host VAX, and the computational result is transferred out of Mass Memory and back to the host memory by means of another DMA. In addition, since all the DIAs can communicate individually with the host VAX, one of them can be instructed to keep data moving between the host and Mass Memory while the others only access Mass Memory while doing their respective calculations, without any VAX Unibus I/O overhead.
- **Graphics:** If seven of the Array Processors are calculating graphics data, one of the DIAs can be driving a graphics display using the same data being updated during the calculations, since all of the DIAs can access the same Mass Memory locations.

3. Porting the Software

The problem that we faced in providing this kind of computing resource to Theorynet users was that the software currently developed for a parallel computing environment using the Aptec DPS2400 and a Vax host was all VAX/VMS based. We were interested in combining the power of the Aptec DPS2400 with the networking capabilities of 4.2 UNIX. To this end, we took on the job of creating such an environment. This task involved porting what we could from VMS to UNIX, and then writing whatever else was necessary. The remainder of this paper will discuss the joys and sorrows of taking a fully developed VMS product and porting it to UNIX 4.2bsd, and to show some examples of implementation programs.

The fact that we had three separate component parts to our parallel processing environment——the VAX host, the DPS2400, and the FPS100s——which did not s hare either a high or low-level programming language or operating system complicated the task of creating a single environment for Theorynet users. This lack of a single environment presents a potential problem to the scientists who are our end users——most of whom have little interest in the area of computer process synchronization.

The initial difficulty we encountered in our efforts to port software from the VAX/VMS environment to UNIX involved difficulties with the varieties of FORTRAN implemented in the two environments. Almost 85% of the code we needed to port from VAX/VMS was written in RAT5, a rational FORTRAN different from the RATFOR that is a part of Berkeley UNIX 4.2. A bit of background on the VMS FORTRAN compiler will help put our task in perspective.

DEC's VMS FORTRAN product seems to support all of the F77 enhancements to the original versions of F66. In addition, the VMS version also supports most of the extensions to the language, thereby making it a powerful, easy-to-use compiler. VMS FORTRAN supports *ENCODE/DECODE* statements, *WHILE* and *FOR* constructs, and also does parsing of *FORMAT* statements (thereby allowing variable subsistution in the *FORMAT* statement). At the time we began our project, the UNIX 4.2bsd FORTRAN——f77——met only the minimum requirements for the F77 standard, which meant that some of the abovementioned amenities weren't implemented.

In the UNIX environment, *ENCODE* and *DECODE* statements could be converted into internal read/writes with relative ease. By using internal read/writes, we found that we could essentially do variable subsistution in a string, and then use that string as a *FORMAT* statement. However, there were some problems that we couldn't easily solve by compiler usage:

- The allowable length for variable, constant, and function names was shorter in the UNIX environment than the 32 character VMS length.
- Almost all the Aptec RAT5 code, and subsequent f77 code contained underscores as part of identifier names.
- The UNIX f77 *DATA* statement didn't allow as many data entries as the VMS version.

Of these three problems, the presence of the underscores in identifier names presented the most difficulty——since we had 10 to 12 thousand lines of RAT5 code to convert, with almost every line having identifiers with multiple underscores in them. Some modification to the compiler seemed called for in this case. Since both the UNIX C and f77 compilers append and prepend underscores to unresolved identifiers (in the case of library references, for example), it was decided to just modify one version of UNIX f77, but not install it as the system compiler (even though a lot of people would have liked to use underscores in names). In the end, only two restrictions ended up surfacing. The first and simplest was that an identifier name could never begin or end with an underscore character. This got around the library problem. We also found that we couldn't use an underscore anywhere in a FORTRAN procedure or function name. If an underscore was used anywhere in a function name, the UNIX f77 compiler always generated incorrect offsets for the return values on the argument pointer stack. We spent a fair amount of time just identifying the problem, and then trying to trace it down. We never did find out why the f77 compiler behaved in this way, and in the end we were forced to go through all the RAT5 code removing underscores from function and procedure names. We ended up with only a few long run-on names, but the code worked much better. Since the lexical portion of f77 does not use UNIX's lex program, modifying the parser to accept underscores as valid symbol characters was a simple matter.

The RAT5 we ported to UNIX was the DECUS[3] version written by B. Kernighan and P.J. Plauger (with subsequent enhancements by the University of Arizona, Lawrence Berkeley Laboratories, and the Institute for Cancer Reseacrch). This RAT5 version (about 6000 lines) is written in RAT5 itself. We began by running the RAT5 source through the VMS RAT5 compiler without the RAT5 f77 switch (since this generated *WHILE* and *FOR* statements), and then moved the resulting raw FORTRAN code over to UNIX.

We now have a TCP/IP package for our VMS system, so file transfer between VMS and UNIX is simple, assuming that you have access to another UNIX system on your network. When we began this project, our original procedure involved mounting a VMS tape as /FOREIGN, with /RECORDSIZE larger than the largest record to be transferred. In VMS, record size can be

3 DECUS is a trademark of Digital Equipment Corporation

found by running *ANALYZE/RMS_FILE* on all files that need transferring. The *CONVERT* utility under VMS was then used on each file to write it out to the tape as a non Files-11 byte stream. When reading the tape in on the UNIX end, *dd* was used. It was involked with matching recordsize (specified with the *dd* flag *cbs=⟨recordsize⟩*) and blocksize (specified by the *dd* flag *ibs=⟨blocksize⟩*) as used to write the tape. The *dd* flag *conv=unblock* was also needed to convert fixed length records to variable length. An edit session on the newly read in files would then strip out all the null characters used to pad out the record size. Once the FORTRAN source was on UNIX, all references to the VMS operating system had to be removed, to be replaced with their exact equivalents written in UNIX FORTRAN. This turned out to be laborious (but not too difficult) and we soon had a RAT5 compiler under UNIX. As a test, we used our new RAT5 compiler to compile RAT5 *itself* (using the original VMS source), and then compared the output to VMS output. They matched byte for byte.

While work on the RAT5 compiler was progressing, another member of the Theorynet staff was involved in designing and writing the UNIX 4.2 driver that would provide the kernel software interface between the VAX host and the Aptec DPS2400.

The first Aptec code we tried porting was the Aptec assembler (CAPAS)—which generates AMD 2901 machine code. This initial conversion was aided somewhat by the fact that all of the code was written in RAT5, and the code's original author was also a UNIX programmer. As a result of this fortuitous coincidence, the RAT5 code for the Aptec assembler had a UNIX-like flavor even though it was a VMS product. With RAT5 in one hand and the CAPAS source in the other, one of our summer student employees was able to port the assembler to UNIX. Again, we verified the assembler by means of a comparison between VMS-generated machine code and UNIX-generated code. In every case, the comparisons were exact, given the same input files.

The CAPAS assembler allowed us to port Aptec DIA microcode into UNIX, make any changes necessary, and (once our kernel driver was done) download the microcode to DIA program source. At this point in time, the linker (used to link several object files together and make a task image file) and a loader (to produce a core image to be downloaded to DIA program source) were still to be written.

There were a number of problems we encountered in the course of this work. Under VMS, when you write a variable length record to disk the file system prepends the number of bytes written—and then strips them out on subsequent reads. The writers of Aptec's compiler took advantage of this fact, and without writing any code they were able to always know how many bytes had been written to disk. Although the problem was not particularly difficult to solve under UNIX, it did take a while to find the source of the problem.

The Aptec DPS2400 can be programmed in one of two ways: When extremely high computational speed is desired, the DPS2400 can be programmed by using microcode. The other method involves writing applications in a language called STAPLE, which was conceived and developed by Aptec. STAPLE embodies many of the syntactic features of both C and FORTRAN, and is an interpreted language. A small STAPLE interpreter written in microcode can reside in each DIA on the system. A programmer writes applications in STAPLE, and compiles using the STAPLE compiler. The object code produced by the compiler is actually composed of pseudocode operands that are downloaded into Mass Memory to be interpreted by the interpreter (the compiler does not produce program source microcode). This type of programming is obviously slower than running pure microcode, but the tradeoffs between execution speed and ease of programming seem to be worth it. The AMD 2901 microinstruction set is probably one of the most complex we had ever seen.

Porting the STAPLE compiler from VMS to UNIX was considerably more involved than the CAPAS assembler. Unlike the CAPAS code, the lexical analysis portion of STAPLE had been done using VMS runtime library utility *TPARSE*. Our plan of attack was to identify all token types and values, and write a symbol table manager to store identifiers in the same manner as VMS did. This avoided our having to modify the LALR tables, and anything else that was VMS dependent.

Once we had a clear idea of token type and value, we were able to write a UNIX lex script that would return the same values as VMS's *TPARSE* utility had. Since lex generates C code, the interface between C and FORTRAN had to be well understood. With over 90 token types, the generated lex code was enormous. A lot of the token identification was done in lex action routines using table lookup and small *case* statements.

The other major bit of UNIX software that had to be written was the symbol table manager. The symbol table manager would allow identifiers to be hashed into a table from lex, and then looked up from the STAPLE FORTRAN code.

4. The 4.2 Kernel Driver

The UNIX 4.2 kernel driver for the Aptec DPS2400 presents some unique differences from typical Unibus device drivers, since the DPS is an intelligent, programmable interface. The driver's main functions are:

- to download a small amount of microcode and execute it in order to generate an interrupt. This is done so that the *dia_probe* routine will work.
- to verify that the requested DIA exists, and check for exclusive use by using the *diaopen* routine.
- to handle all interaction from the user level through *ioctl* calls. There are no *dia_read* or *dia_write* entry points in this driver.
- to download an executable (core image) kernel into DPS program source and start it running.
- to make a call to the DPS kernel on behalf of a user in order to start an application running. Application microcode is actually a part of the kernel.
- to watchdog time the DIA application for a user-specified amount of time.
- to oversee the DMA transfers between Mass Memory and user space (even though the DIA is considered to the the initiator and controller of the DMA transfers).

Each DIA in the system can run a different version of microcode kernel. Therefore, there is a table in each microcode core image stating what subroutines it contains. It is the driver's responsibility to query the DIA kernel in order to verify that the requested subroutime is present.

Once a call is made to a DIA microcode routine (APSLAVE in our case), the DIA can sever its communications link with the VAX host. At this point the DIA is said to be in the *DIB mode,* and it can only communicate with its associated Unibus device (Array Processor). The DIA does not return a success code to the DIA driver until after the DIA microcode routine successfully completes. This period of time until successful completion could be days weeks, or even months. The user can specify an execution time limit, and if this time limit expires the DIA driver will forcefully terminate the running DIA.

This non-return by the DIA to the UNIX kernel driver can be handled at the user level by forking a child process for each open DIA (up to 9 on our system). There is still a multiple process scheduling and resource overhead problem that has to be handled by the UNIX operating system.

The remainder of the paper will discuss some actual programs that exercise this configuration, discuss some of the applications currently running or planned to run with the parallel Array Processors, and present some ways of making the user interface more attractive to general users.

At the time this paper was written, there were numerous test programs written designed to test the functionality of the system. The program included in the first appendix to this paper was written during the process of debugging the system. In essence, the test program loads data into an Array Processor (in this case, the data loaded is two vectors). The Array Processor then writes the data into Mass Memory and sets a synchronization bit in Mass Memory that all the other DIAs are waiting on. When the synchronization bit is set, all eight Array Processors will read the same data into their respective main data memories, perform a vector add (VADD) on the 2 vectors, and disconnect from the DIA and re-connect to the host program running on the VAX. The host program will then use the Array Processor software directly to read the results and perform the comparisons necessary to verify that each Array Processor received the correct initial data.

Although this program is a trivial example, it can easily be seen how the 2 vectors can be expanded into more complex data. In a similar sense, the trivial VADDs can be expanded into large parallel Array Processor applications.

One of the current programs is used for nearest-neighbor lattice gauge theory calculations of interest to theorists working in the field of Quantum Chromodynamics (QCD). This program[4] uses FORTRAN, and provides us with something of a base against which we can measure the potential speed to be gained by vectorization and parallelization.

Some preliminary timings show that if the communications between the Array Processor and the DIAs occur once per second and 15625 4 byte words are transferred, then computing efficiency of the system is about 93% (the remaining 7% is communications overhead). The overhead on Array Processor-Mass Memory I/O is about 0.0005 sec. (on a mix of 5 system calls to *TCMMIO* and one to *DOMMIO* and *WTMMIO* each[5]).

5. The Programming Environment

One high-level language Array Processor compiler presently available is the APTRAN compiler, written by Kenneth Wilson and Donna Bergmark at Cornell. This is the first compiler written for the Array Processors, and predates the APFTN and the Toast-FORTRAN compilers marketed by Floating Point Systems. The APTRAN compiler supports a subset of FORTRAN66, and so is at times rather painful to use. This relative inconvenience must be weighed against having to write programs in Array Processor assembly language (APAL), however.

The FPS100s are driven by software from APUNIX Inc. This software allows for programming the Array Processors by means of 2 Vector Function Chainer (VFC) compilers—one a C-like VFC, and one a FORTRAN-like VFC. These compilers are useful for for programs that can be written entirely as a sequence of Math Library calls[6], for programs that are hand-coded in APAL, or for testing purposes.

---

4 This sample Monte Carlo calculation is included in the Theorynet document **A Guide to Parallel Processing on the Aptec-FPS100 System.**

5 See **A Guide to Parallel Processing on the Aptec-FPS100 System** for more information on these Array Processor to DIA system calls.

6 The math library routines available to Array Processor users range from very simple operations (such as multiplying all the elements of a vector by a scalar) to more complex mathematical operations (calculating the eigenvectors and eigenfunctions of a matrix). In practice, it is possible to achieve a significant fraction (maybe about 0.5) of the peak performance if the computationally intensive parts of the program can be written as a sequence of math library calls.

At present, the high-level language APTRAN compiler resides on the IBM 3081. Our users have access to an IBM exec *APCOMP* that compiles, assembles, links, and edits the relocatable AP object code and the Host-AP Software Interface file (HASI) necessary for use in VAX—FPS100 environment.

6. Some Possible Enhancements

The system described here offers us a considerable improvement in computational speed at a fraction of the cost of some alternative options. It also has the advantage of being expandable (2 DPS2400s can be connected together) to suit almost any application, should the need arise. There are some enhancements that would make our current system even more useful.

For our future simulations (and some of our current projects as well), it rapidly becomes obvious that 4 MBytes of Mass Memory may not be enough for our needs. A number of simulation problems fall easily into the Gigabyte range in terms of Mass Memory requirements. If this data must be transferred from the VAX host, then there are still plenty of Unibus I/O bottlenecks. One possible solution is to put a large disk farm on the DPS2400 in order to keep it running at its maximum rate. To this end Aptec has built another interface like the DIA called a Data Interchange Processor (DIP). This interface is functionally like a DIA, but it is faster. Aptec has also built a controller board that will allow the DIP to interface to an IBIS[7] 1.2 GByte disk (The IBIS is used extensivly on CRAY machines). The interface bus between the IBIS and DIP runs at 12 Mbyte/sec., with up to 4 IBIS disks able to hang off the controller. Porting the DIP microcode to UNIX presents some problems in that the new AMD290x microprocessors use a slightly different microinstruction word, so our CAPAS assembler cannot be used. Also, there are a lot of VMS Files-11 dependencies coded into the existing DIP microcode disk driver which would preclude simply moving the microcode into the UNIX environment.

A second possible enhancement would involve writing a TCP/IP kernel that would run in one of the DIAs, and thus provide another way of easily and quickly getting large amounts of data into Mass Memory.

Future work in the area of parallel array processor systems will likely be in the areas of user interface design and parallel compilers. The LESTAP language developed at O.N.E.R.A.[8] is one current effort in the area of parallel compilers.

---

7 IBIS is a trademark of IBIS Systems, Inc.

8 see *Implementation of Parallel Numerical Algorithms on a Multi-AP and Shareable Mass Memory System* by P. Leca (published in **Proceedings of the 1985 Array Conference**

APPENDIX A

Test Program

```
/*
 *  This is a test program designed to exercise the
 *  functionallity of the DEVVAX parallel array processor
 *  system.
 *  Rich Jaenson
 *  Cornell University
 *  Theory Center
 *  162 Clark Hall
 *  Ithaca, New York 14853
 *  (607)-256-4981
 */
/* to compile... cc mdia4.c -lm -lFPS  -lCDPS */

#include <stdio.h>
#include <sys/types.h>
#include <math.h>
#include "/sys/DEVVAX/dia.h"
#include <sys/diareg.h>
#include <errno.h>
#include <sys/ioctl.h>
#include <sys/file.h>
#include <sys/wait.h>
#include <signal.h>
main()
{
#define SIZE 1000
#define NO_BUF 64
    int one();          /* declare signal handler */
    int fdcnt;               /* used by children to close all
                              * inherited file descriptors.
                              */
    long time();
    long tloc_b,tloc_e;
    double sqrt();
    extern fps_no;  /* fps_no is declared in APEX */
    u_short apnum,action;
    short status;
    float a[SIZE],b[SIZE];   /* our two test vectors */
    float result[9][SIZE];   /* used to store results at test end */
    float xmpl;
    char s,t;
    int diafd;
    int ia,ib,ic,id,ifmt,i,j,istat;
    int pid;
    union wait *child;
    char *tskname = "APSLAVE";  /* name of dia micro-code routine */
    int ap_count=0;           /* count of avail AP's */
    u_short ap_num[9];          /* AP number of avail AP */
    char dia_name[9][10];       /* array of built dia names */
    u_short rjap[9];
    int argcnt,waittime;
    caddr_t argbuf[],retbuf[];          /* arguments for calldia() */
    argcnt = 0;
    waittime = 1;

/*
 * Because of the interactions between APEX controlling
 * the AP's and also the DIA's controlling the AP's we
 * found that if we did not call exit() explicitly on a
```

```
 * (ctl)c there was a race condition as to who caught the
 * signal first and resources (unibus buffers) were not
 * getting released properly.
 */
   signal(SIGINT,one);
   tloc_b = time(0);
   printf("begin time=%d0,tloc_b);
/*
 *  The two vectors this test program adds are
 *  each SIZE long.  One contains an integer
 *  and the other its square root.
 */
    for (i=1;i <= SIZE; i++){
         b[i] = i;
         a[i] = sqrt((double)i);
         }
/*
 *  This next bit of code goes out and looks
 *  for any available AP's to run on.  In the
 *  array ap_num[] will be stored the AP numbers
 *  1-8 of any available devices.
 */
   action = 1;              /* don't wait for AP to become avail */
                            /* i.e. just return its current status */
   for ( apnum=1; apnum <= 8; apnum++){
     apinit(&apnum,&action,&status);
     if ( status < 0)       /* AP is not available */
         continue;
     else {                 /* AP is available */
         ap_num[++ap_count] = apnum;
         aprlse(); /* here we are just looking for available
                    * AP's and don't want anything else done
                    */
         }
   }
   if ( !ap_count > 0 ) {
      printf("There are no AP's available..exiting0);
      exit();
      }
   printf("there are %d ap's avail to test...",ap_count);

   /*
    *  Now that we know which AP's are available
    *  use apsel to select and init AP's.
    *  Apsel counts from 0->MAXAP-1 whereas apinit
    *  counts from 1->MAXAP.
    */
   action = 0;              /* this forces apinit to wait for AP... */
                            /* shouldn't happen since we know which */
                            /* AP's are avail. */
   for ( i=1; i <= ap_count; i++){
      rjap[i] = (ap_num[i] - 1);   /* adjust ap num by -1 for apsel */
      apsel(&rjap[i]);              /* apsel() allows random selection */
      apinit(&ap_num[i],&action,&status);
      apidis();             /* disable software interrupts */
                                    /* the APTEC will control these */
/*
 * The APTEC uses interrupt bit 5 (Interrupt Host Enable) to
 * synchronize with the AP.  Therefore this bit HAS to be
 * clear before starting up a DIA.  note: The front panel
 * lites on the FPS-100 is an 'or' of all interrupt status
 * and the int lite will come on after apidis() clears it
 * but infact the IHE interrupt is still off.  Took a while
 * to figure that one out.    ....rich
 */
```

```
      setapm(NO_BUF);                  /* Take AP out of auto-sync so  */
                                       /* that apwr() and apwd()'s work */
      sprintf(dia_name[i],"/dev/dia%d",status);
/*
 * The above sprintf builds the /dev/diaX name based on the
 * AP we just grabbed.
 */
      printf("dia_name = %s0,dia_name[i]);
      }

/*
 *  Select the first AP and load our 2 vectors into
 *  its  APMD. When the first ap is started (via apinitvecs)
 *  it will copy these vectors into MM.
 */
   apsel(&rjap[1]);          /* select the first ap */
   ifmt = 2;                 /* this is floating point type 2(FPS) */
   ia = 0;                   /* first vector will go to APMD addr 0 */
   ib = ia + SIZE;     /* second vec will go to APMD addr 1000 */
   ic = ib + SIZE;
   id = 4000;
   apput(a,ia,SIZE,ifmt);  /* use APEX to load both vectors into */
   apput(b,ib,SIZE,ifmt);  /* AP Main Data Memory */
   apwd();                      /* wait until DMA is finished */


   apinitvecs();   /* this is the vfc prog to */
                           /* load ap vectors into mm */
                           /* this is only done from the first AP */
   apwd();
   printf("ap %d..back from apinitvecs..0,ap_num[i]);

/*
 *  Now go and start all the other AP's by first
 *  selecting them and then calling childvecs()
 *  which is a VFC program.
 */
   for( i=2; i <= ap_count; i++){
      apsel(&rjap[i]);                  /* select the next ap */
      if (i == ap_count) {   /* i.e. this is the last ap */
        childvecs();              /* future use....  */
          /* This type of routine can be used to load different
           * versions of code into the AP's depending on
           * synchronization or calculation needs.  Orig. intended
           * to load code into the last AP stating that it was
           * the last and it would set a "sync" bit allowing
           * all the other AP/DIA's to start at the same time,
           * but the ascertation as to who was last was more dependent
           * on the order that the children started their DIA's.
           */
        apwd();
        }
      else {
        childvecs();
        apwd();
        }
      printf("ap %d ..back from childvecs..0,ap_num[i]);
    }
/*
 *  The parent will fork up to ap_count copies
 *  of itself and then wait for the children
 *  to expire.  Each child will open a dia, call
 *  diainit, and then start a copy of APSLAVE
 *  in its associated dia.  APSLAVE is the dia
 *  micro-code that will handshake with the AP
```

```
 *  and do all data movement between MM and APMD.
 *  calldia() roadblocks until it has received a
 *  apdone from the AP, which is the last instruction
 *  in all the VFC programs.
 */
    for ( i=1; i <= ap_count; i++){
          if ( fork() != 0) {    /* i.e. parent */
                    continue; /* go fork another child */
          }
/*
 * Ici nous sommes l'enfant..
 */
          else {

          /* close all file descriptors inherited from parent */
          for (fdcnt=3; fdcnt <= 32; fdcnt++)
             (void) close(fdcnt);

                    diafd = open (dia_name[i],O_RDWR,0);
                      if (diafd < 0){
                        perror("child");
                        exit(-1);
                      }
                    /* diainit causes the micro-code image
                     * that contains APSLAVE to be downloaded
                     * into the associated DIA.
                     */
                    if (diainit(ap_num[i],diafd) < 0){
                              prterr(ap_num[i]);
                              exit(-1);
                    }

                    /* this is the call to APSLAVE */

                    if ( calldia(diafd,tskname,argcnt,argbuf,
                              retbuf,waittime) < 0 ) {
                                        prterr(ap_num[i]);
                                        exit();
                              }
                    if (close(diafd) < 0) {
                        perror("child");
                        exit(-1);
                        }
                    printf("CHILD DONE0);
                    exit(0);  /* successful child exit  */
             }     /* end of else  */
          } /* end of for fork loop */
/*
 *  this is the bottom of parent code.
 *  The parent will wait until all of
 *  its children have exited before continuing.
 */
          printf("PARENT...waiting for all children to exit0);
          for (j=1; j <= ap_count; j++){
                    pid = wait(&child);
                    }
/*
 * At this point all AP's are done so go get
 * results
 */
  printf("EVERYBODY is fini...");

  for ( i=1; i <= ap_count; i++) {
    apsel(&rjap[i]);
    if ( i == 1 ) {  /* if first AP the results are stored */
```

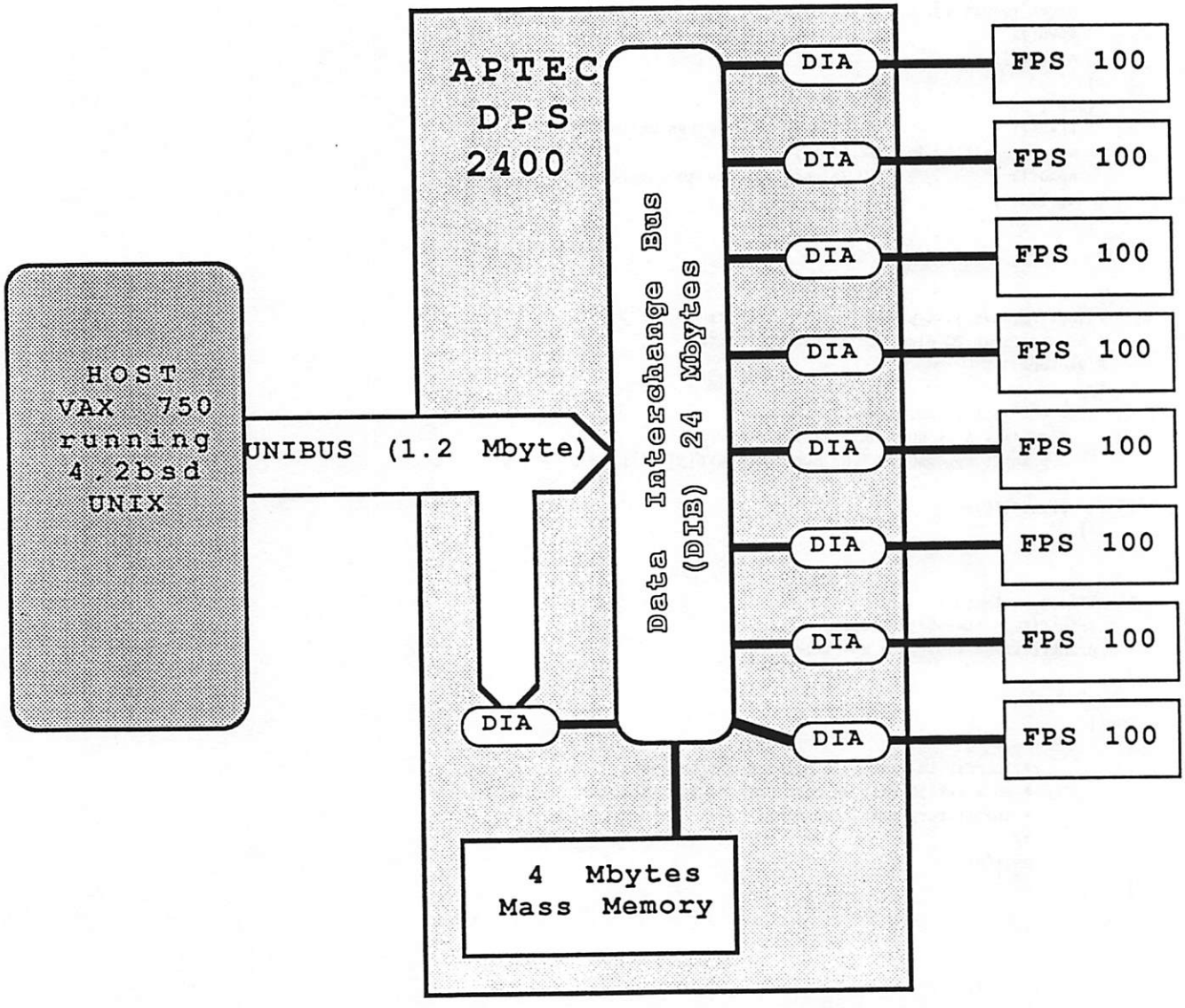
```
                        /* in a different location in the AP  */
      apwr();
      apget(result[i],id,SIZE,ifmt);
      apwd();
      aprlse();
    }
    else {
      apwr();                  /* make sure AP has halted */
      apget(result[i],ic,SIZE,ifmt);
      apwd();                  /* wait for dma to complete */
      aprlse();
    }
  }

  /*
   * what follows just prints out the results
   * of the first 20 elements of the calculated
   * vectors.
   */
     for ( i=1; i <= 20; i++) {
       for (j=1; j <= ap_count; j++) {
         printf("ap%d=%6.3f ",ap_num[j],result[j][i]);
       }
       printf("0);
     }


    tloc_e = time(0);
    printf("end time=%d0,tloc_e);
    printf("time= %d0,(tloc_e - tloc_b));
}

one()
{
          /* inorder to make sure nothing got hung-up
           * on a <ctl>C exit we made sure we went through
           * the library exit() routine.
           */
           exit();
}
```

Illustration 1: A Schematic Diagram of the VAX/Aptec/FPS100 System



A Stable Storage Package

Bruce Ellis[1]

Basser Department of Computer Science
The University of Sydney, N.S.W. Australia 2006

ABSTRACT

Many systems applications maintain structured disk files that should be resilient to machine crashes and abnormal software events. A stable storage package maintains a disk file data structure that ensures such a property. This paper describes a stable storage package for the UNIX† operating system. This package requires no special kernel support and is designed to take advantage of the disk buffer cache rather than fight against it. It provides a means for automatically restoring a structured file to the state before an abnormal software termination, or to a state consistent with some short time before a machine outage.

1. Stable Storage

A stable storage package provides a means of maintaining the consistency of structured disk files in the event of machine crashes and abnormal software events. In the strictest scheme such a package guarantees a client that the effects of a transaction are *stable* (i.e. will not be effected by an abnormal event) when indicating success to the client. This guarantee of transaction is essential to many database applications. For example a client might require that a return from the operation "commit transaction" indicates that the database modification is stable.

Such a stable storage scheme cannot readily co-exist with the UNIX buffer cache. Although this cache preserves the semantics of (logical) I/O operations under normal conditions, this is not the case in the event of a system crash. In particular the ordering of modifications is not preserved and the return to a client up to some minutes before a system crash is not a file-system transaction guarantee.

We implement a weaker stable storage scheme that works well in this environment.

2. The UNIX Buffer Cache

The UNIX buffer cache is a collection of in-core copies of disk blocks and auxillary data structures in the UNIX kernel. Its purpose is to cache the reading of disk blocks and delay the writing of disk blocks, in the hope that the block will be logically rewritten or discarded before it need be physically written. This is in the same spirit as a CPU's main memory cache.

The cache is effective. It is known to increase logical I/O rates considerably. A ratio of ten to one between logical and physical I/O operations is not unknown.

Traditional stable storage schemes are not applicable since a logical write may not make it to disk for quite some time (in the order of several minutes) and the ordering of disk block modifications is not preserved (even the disk drivers are free to re-order I/O operations).

---

1 Work on this project is funded by argc, The Australian Research Grants Committee.

† UNIX is a Trademark of Bell Laboratories.

The problem of implementing a stable storage package for UNIX can be addressed in several ways.

- adopt a pessimistic view of logical writes, i.e. that a logical write takes place after some long period of time. (The UNIX program *update(8)* periodically flushes sandbagged writes using the system call *sync(2).*) This approach is absurd, it treats a UNIX file-system as an extremely slow writer. Stable storage updates in this scheme take a very long time.
- circumvent buffer cache by using a special file system on a raw device. This approach loses the benefits of the buffer cache. Known stable storage strategies for uncached file-systems can be employed in this case. This scheme is not manageable in all but very specialised user environments.
- provide kernel support for forcing I/Os e.g. the 4.2bsd *fsync(2)* system call. Such a scheme is not portable and to a large extent loses the benefits of the buffer cache. (The *fsync(2)* call is also known to degrade system performance severely. A files indirect blocks are read to determine which blocks belonging to the file need to be flushed from the buffer cache. For all but very small files this involves a large amount of I/O and in effect a flush of the buffer cache.)

We adopt a different approach. Rather than making guarantees about transactions we implement a stable file policy. This policy guarantees that the file structure is maintained in the event of a software crash, and that in the event of a system crash the file structure is in a state consistent with some short time before the crash. Such a policy is adequate for many tasks.

For example, this stable storage package was designed to be used by a distributed name server that maintains a list of names and their property lists. At each host the name server maintains a stable file containing this data and index files (which are themselves stable files) for quick access to the data. In the event of premature termination of a process modifying these files, a consistent data structure should be maintained. The fault may have been in the client or the machine. This task does not require stable transaction guarantees, merely guarantees of file structure consistency.

3. Client Interface

The client interface is straightforward. We provide primitives on which much more complex systems can be based. The interface can be divided into four classes of routines.

i) routines for creating, opening and closing stable files. The create and open routines return a pointer to an *sfile* structure which is passed to all subsequent calls to the package. A stable file can be opened for read or read/write.

ii) routines for dealing with records within a stable file. Records are of variable length and are denoted by an integer, much like the *inode* number of a UNIX file. Routines are provided for reading records, locking and reading and subsequently releasing or rewriting records, and for creation and deletion of records.

iii) routines for storing and retrieving information to be associated with a stable file. These routines maintain a limited amount of storage for attribute/value pairs. They provide the operations of assignment, deletion and retrieval. Such routines are useful for tagging and identifying stable files.

iv) routines for maintaining and repairing stable files. These routines will not be discussed in this paper.

4. Principles

The stable storage package is based on simple principles.

Several instances of each record are kept. These instances and the data blocks (and indirect blocks) are time-stamped. When a record is updated an instance is discarded and replaced with one corresponding to the new data. The discard algorithm maintains exactly one instance that is known to be "clean", because it is older than the I/O guarantee period. The algorithm attempts to minimise the data that would be lost in the event of a crash.

The I/O guarantee period is the *update(8)* cycle length plus the maximum driver latency time. A pessimistic measure of the latter is easily obtained from the size of the buffer cache, the maximum disk seek and transfer times.

An incompletely written instance (caused by a system crash) can be detected by checking time-stamps. Recovery after a crash is automatic and relatively straight forward. Performing such recovery is matter of course for all package operations.

The file operations involved in implementing each of the package's functions are ordered so that a software failure will not undermine the data consistency. In the event of such a failure some data blocks may be allocated and not used. These will be picked up by a subsequent garbage collection.

As an example, here is an outline of the ordering for a record update operation:

```
lock file

allocate data blocks from map
rewrite map
force lru package writes

write data
force lru package writes

if instance to delete

    remember instance
    update inode
    force lru package writes

    update map marking old data blocks as free
    rewrite map
    force lru package writes

    mark freed pages as free

else

    update inode

force lru package writes
unlock file
```

5. File Design

The UNIX file that implements a stable storage file consists of four parts. The first block contains a magic number and the attribute/value data. The second part contains a *map* of the rest of the file, two bits per block, each block being classified as one of

i) unassigned block

ii) inode block containing at least one free inode (an inode describes a record, the instances of its data, analogous to a UNIX file system inode)

iii) inode block with no free inodes

iv) data block

The third part contains a record of current users and record locks. This is used to implement a FIFO record locking policy.

The fourth part is the inode and data blocks as described by the map. As well as being time-stamped each block of this part contains a tag which should correspond to the map's tag. This allows the automatic detection and recovery from incomplete transactions.

6. Implementation

The stable storage package runs on top of an LRU buffer package that arranges file locking and eliminates redundant I/O operations. This file locking is done in either a UNIX version specific manner, or by a portable but less efficient method using lock files.

Record locking is achieved using data structures maintained in the third part of the file (the record of users and locks). The file is locked only during modification, i.e. the "read with lock" operation does not leave the file readlocked.

User and lock structures are also timestamped. Automatic user corpse and stale lock detection and elimination is performed as part of each operation. Deadlock detection and elimination could be easily provided. In the current application (the name server) this is not required because of the client locking protocol.

The correct choice of an instance discard algorithm is not clear. If the interval between record updates is greater than the I/O guarantee time divided by the number of instances kept, the choice need never be made — there will always be a "clean" instance which can be discarded. If this is not the case the algorithm should choose probabilistically, weighting instances as some function of their age and perhaps take into account the history of intervals between record updates. Newer instances are more desirable, older instances are more likely to have made it to disk. Betting on the "valuable" new instance is speculative, on the more stable but less enticing instances is blue-chip.

It is not easy to determine a good repair policy. If a client detects an inconsistency how much time should be put into repairing the file, enough to merely complete the clients transaction or to complete clean the record in question or even the whole file? Who should pay for garbage collection? Debugging of the package is certainly complicated by the automatic repair — we must safeguard against routines trying to repair non-existent inconsistencies, or covering up the tracks of a beserk routine!

7. Conclusion

We have implemented a stable storage package which takes advantage of the UNIX buffer cache rather than fighting against it. To its clients this package is essentially a self-repairing file-system.

Dbxtool

A Window-Based Symbolic Debugger for Sun Workstations

Evan Adams & Steven S. Muchnick

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043

1. Introduction

Dbxtool is a window- and mouse-based interactive debugger for C, Pascal and FORTRAN programs which increases programmer productivity by extending the facilities of the *dbx* debugger (developed at the University of California at Berkeley) and improving the user interface through the use of SunWindows*, the Sun Microsystems window system. By using the mouse instead of the keyboard as the primary input mechanism, *dbxtool* eliminates the need to type variables, line numbers, breakpoints, and most commands. *Dbxtool*'s windows provide a view of the program being debugged and detailed information about the state of the program, thus affording two qualitatively different perspectives on the debugging problem. *Dbxtool* (and the underlying *dbx*) have also been extended to make them much more powerful debugging tools. The new capabilities include the ability to debug multiple-process programs, processes which are not children of the debugger, and the 4.2 BSD UNIX† kernel.

2. Historical Background

Dbx is descended from *pdx*, a debugger written by Mark Linton for the Berkeley Pascal interpreter in the fall of 1981. In fall 1982 a version capable of debugging compiled C code on the VAX (and the first to be named *dbx*) was installed at Berkeley. In spring 1983, *dbx* was distributed as part of the 4.1c BSD release of Berkeley UNIX and was extended by Alastair Fyfe to support debugging of FORTRAN 77 programs. Also, that spring it was ported by Linton to 4.1c on the Sun workstation and in the fall of 1983 it was released as part of 4.2 BSD.

In 1984, *dbx* was extended anew at Sun to handle debugging of FORTRAN 77 and Pascal programs, several extensions were added to its command language and many, many bugs were fixed in it. Once *dbx* had become a reasonably stable and mature glass teletype-oriented debugger for simple programs written in any of Sun's supported languages, the next step was to improve its user interface and to make it a full-featured debugger for arbitrary programs. We were convinced that one could replace much of the typing *dbx* requires by pointing and that a single dialogue area inexorably scrolling upward was not the best use of screen space. We also confronted the lack of good facilities for debugging multiple-process programs and lack of good support for debugging the UNIX kernel.

---

*SunWindows is a trademark of Sun Microsystems.
†UNIX is a trademark of AT&T Bell Laboratories.

3. Improving the User Interface

Given a glass teletype-oriented debugger running on a bit-mapped workstation with a powerful window system, the obvious next step is to build a more effective user interface. What is not so obvious is how the user interface should look. Some issues were clear to us:

- There should be a scrolling window to display the source text of the program being debugged.
- There should be a mechanism for selecting commands and their arguments with the mouse.
- There should be graphical feedback of the locus of execution and the current breakpoints.
- There should be some sort of mechanism for displaying the current values of variables as execution proceeds.

On the other hand there were also many questions.

- Should the source window be part of the debugger or should it be an arbitrary window that the user can associate with the debugger?
- Which of the many available commands should be constructible with the mouse? If not all, what other mechanism should be provided for constructing them? Should the user be able to tailor the set to his individual needs or style?
- What should the details of the variable display be? Is it important to display data structures graphically?
- Is there a need for a command dialogue (like the traditional mode of interaction with *dbx*) within *dbxtool*?

To settle these issues we built on the experience of the designers of previous window-based debuggers and on the debugging habits of software engineers at Sun. We instrumented a version of *dbx* to report the usage of individual commands. Six commands stood out as the most frequently executed.

We felt that the most commonly used commands should be constructible with the mouse but that the choice of those commands should not be frozen by the system — the user should be allowed to define his own. We decided the source window should be part of the debugger to facilitate annotating the source code with symbols like the stop sign used to denote a breakpoint and the right arrow used to denote the next line to be executed. We decided the variable display should resemble the output of *dbx*'s **print** command and that graphical display of data structures, while desirable, was not feasible at this time. We also decided that a command dialogue area was necessary, as *dbx* provides many more commands than could reasonably be attached to buttons on the screen, some with rather esoteric argument sequences.

3.1. The *Dbxtool* User Interface

An invocation of *dbxtool* consists of five subwindows: 1) a status window, 2) a source window, 3) a menu of command buttons, 4) a command dialogue window, and 5) a variable values display window (see Figure 1). The status window displays the file and line number range of the code in the source window and information about the current state of the debugging process. The source window usually displays the current focus of execution, though it can be moved under user control by scrolling or regular-expression string searching to any part of a source file

(or to any other file). The buttons window contains a menu of commands that can be constructed with the mouse. The command window provides a command dialogue area where the user can type commands and where the commands invoked from the buttons window are echoed. The variable values display window (generally called the ''display window'') displays the values of selected variables and expressions whenever execution halts. The source, command and variable display windows all have scroll bars, so the user can easily move his focus of attention within the displayed material.

By default, the menu of command buttons contains the six most commonly used commands — **print** (prints the value of a variable or expression), **next** (single steps by source line and skips over function calls), **step** (single steps by source line and steps into function calls), **stop at** (sets a breakpoint at a source line), **cont** (continues execution until the next breakpoint is encountered), and **stop in** (sets a breakpoint at the beginning of function or procedure), plus a **redo** (issue a command again) button.

Commands are constructed by making a selection with the mouse and clicking a command button. The mechanism for specifying a command button consists of a string and a *selection interpretation*. The selection interpretation specifies a (built-in) function which takes the selection as its argument and returns a string. The *dbxtool* user interface appends the string to the button's string to construct a command and passes it to the debugger. There are five selection interpretations:

literal	A selection may be interpreted as representing exactly the selected material.
expand	A selection may be interpreted as representing exactly the selected material, with the exception that it will be expanded if either the first or last character of the selection is an alphanumeric character or underscore. It will be expanded to the longest enclosing sequence of alphanumeric characters or underscores.
lineno	A selection in the *source* window may be interpreted as representing the (line number of the) first source line containing all or some of the selection.
command	A selection in the *command* window may be interpreted as representing the command containing the selection.
ignore	The selection may be ignored.

The **print** and **stop in** buttons use the the *expand* selection interpretation. Thus, the value of a variable may be displayed by selecting as little as one character of its name in the source window and clicking the **print** button. The **stop at** button uses the *lineno* selection interpretation, so a breakpoint may be set at a line by selecting a single character anywhere within the line and clicking the **stop at** button. The **next, step** and **cont** buttons use the *ignore* selection interpretation as none of these commands require any arguments. The **redo** button uses the *command* selection interpretation.

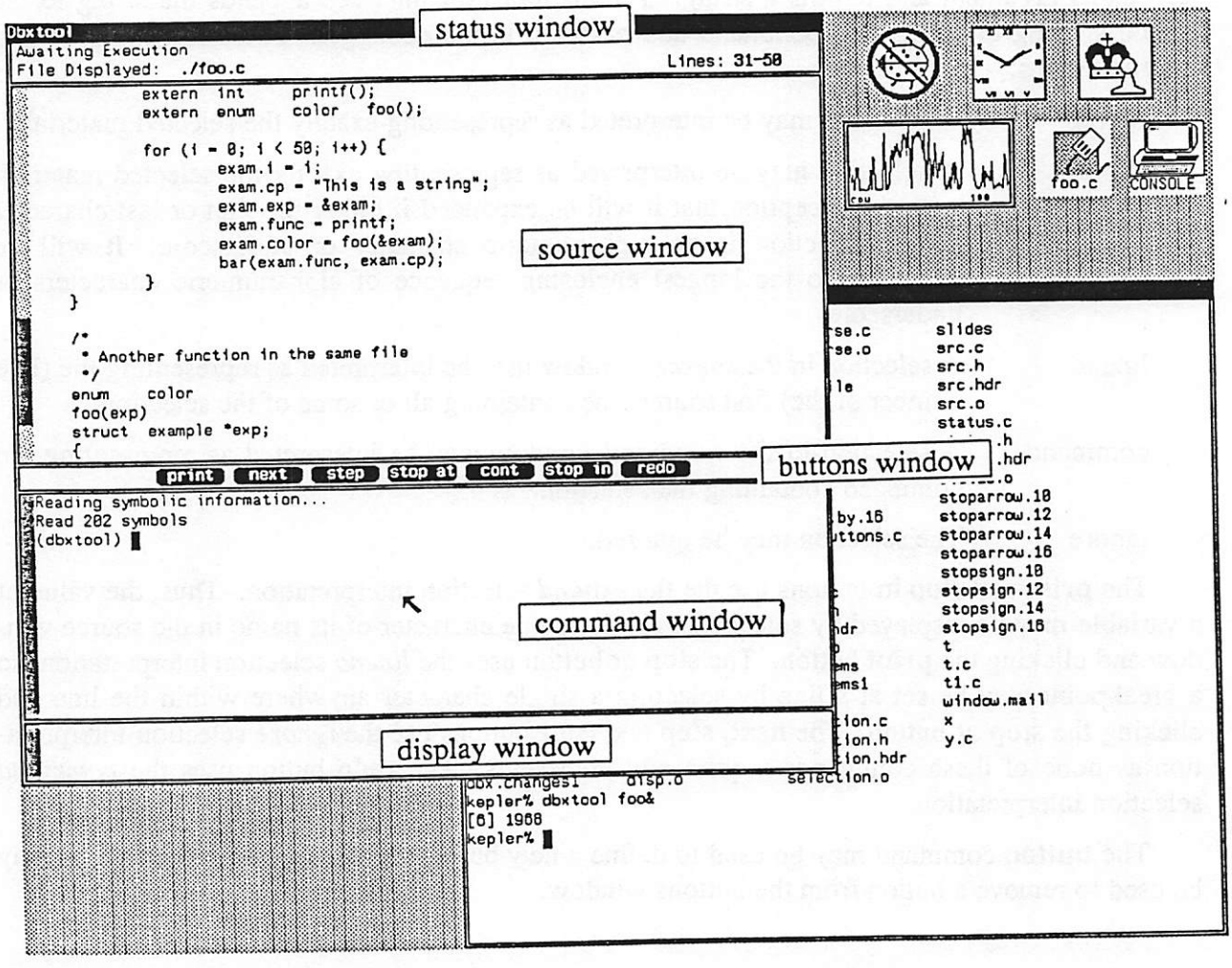
The **button** command may be used to define a new button and the **unbutton** command may be used to remove a button from the buttons window.

4. *Dbxtool* Walkthrough

This section presents a short *dbxtool* session to provide the flavor of its usage. The figures are screendumps from a Sun workstation.

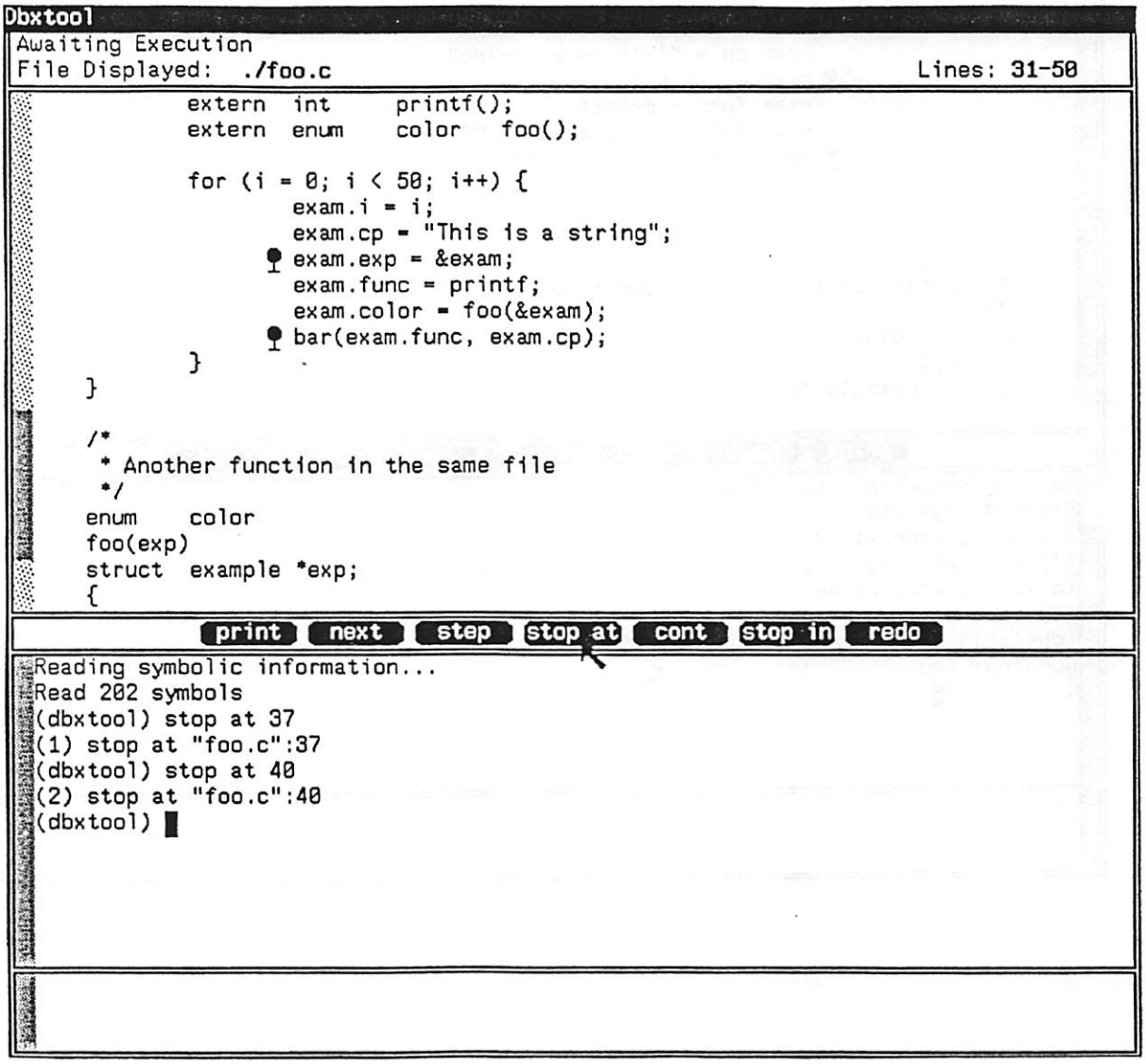
Figure 1 shows an invocation of *dbxtool* on the program **foo**. The status window indicates that the program has not yet begun execution and shows which lines of which file are currently displayed in the source window. The source window shows source text from the **main** procedure and the command window is ready to accept commands.

Figure 1



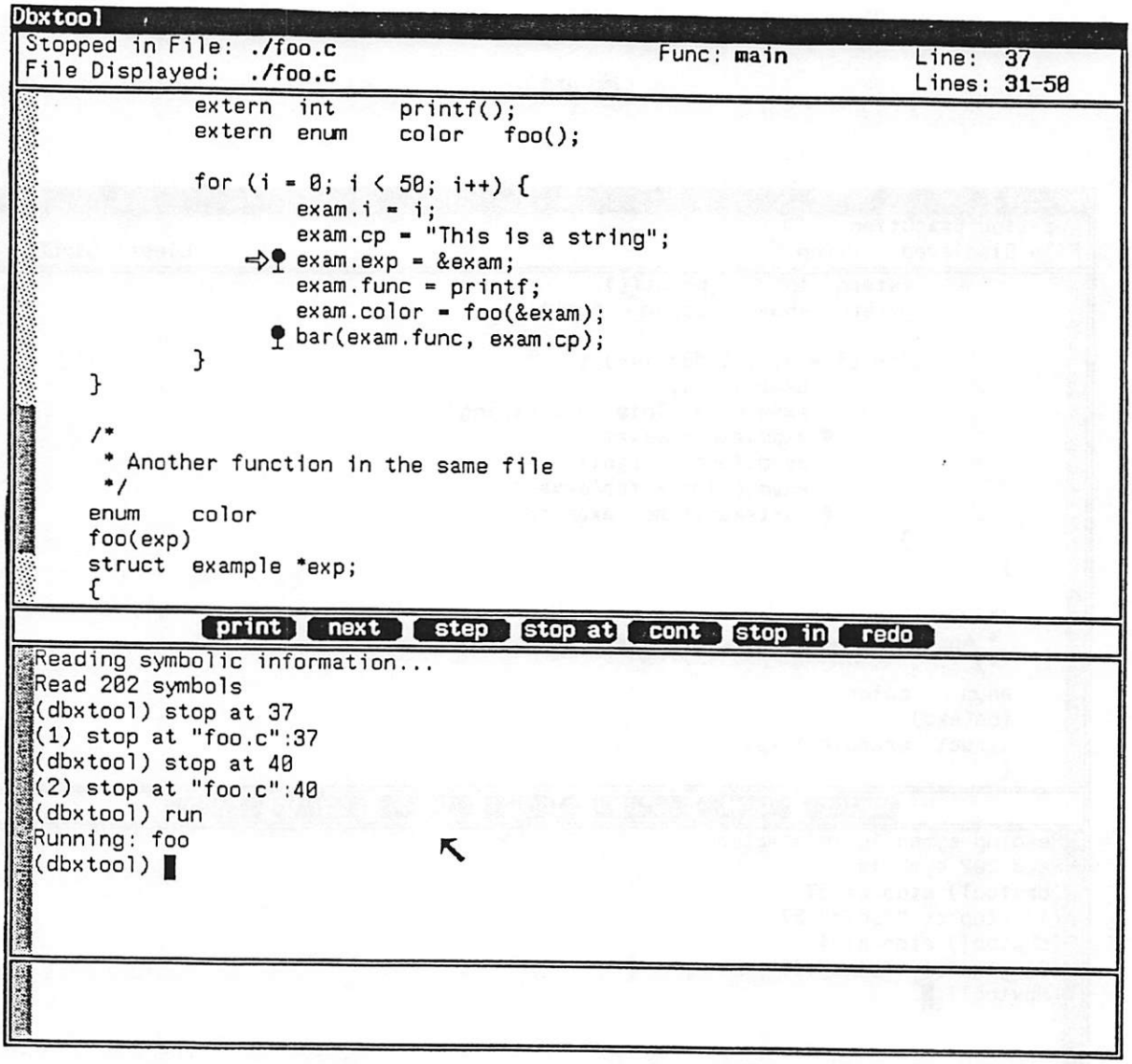
In Figure 2, the user has set two breakpoints using the mouse. A selection is made in the line beginning "exam.exp" (line 37) and the **stop at** button is clicked. *Dbxtool* displays a stop sign next to that line and the constructed command is echoed in the command window followed by the debugger's response. A second breakpoint is then set on the line which invokes procedure "bar".

Figure 2



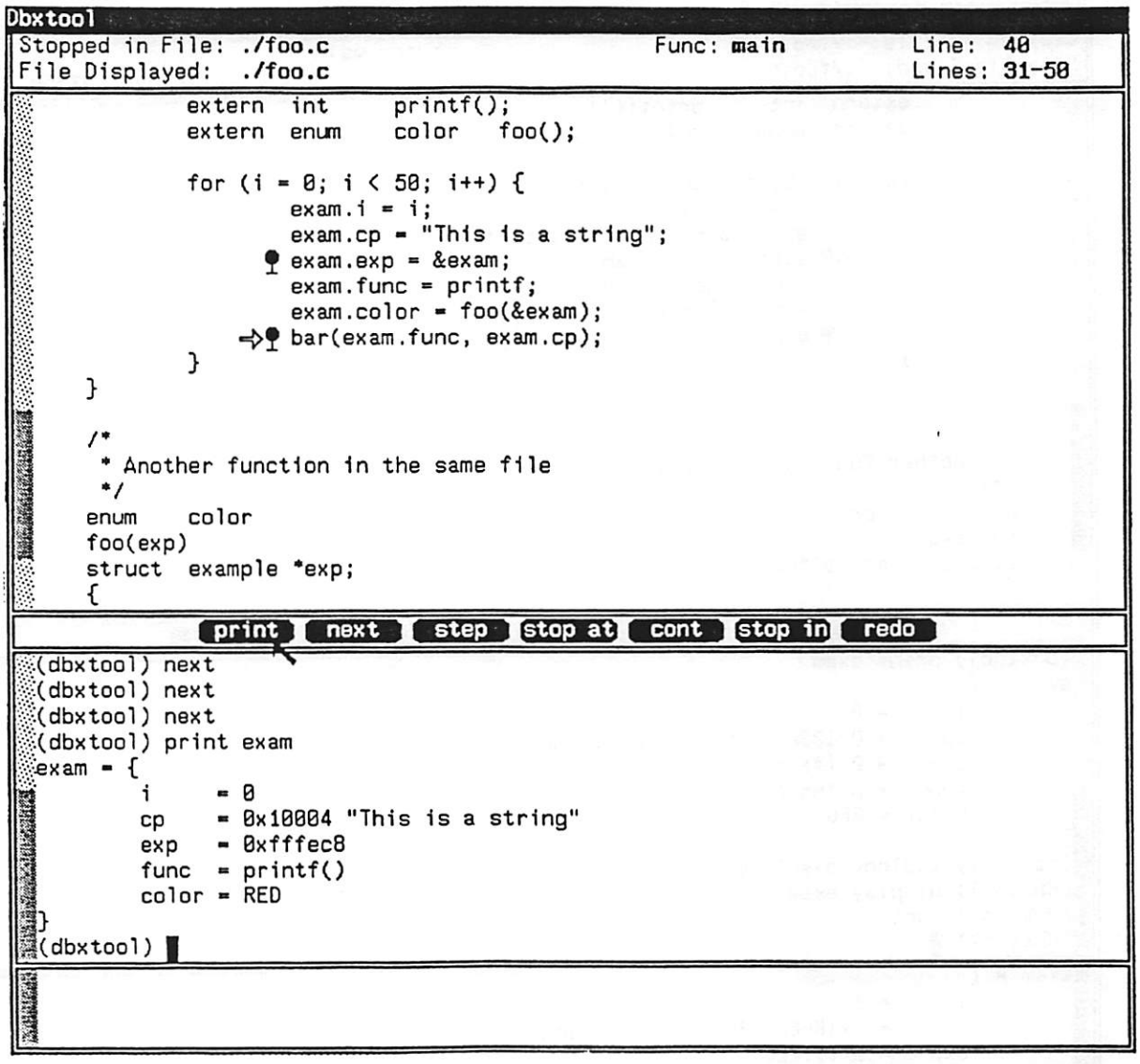
In Figure 3, the user has typed the **run** command into the command window. Execution of **foo** is initiated and it continues until the first breakpoint is encountered. *Dbxtool* then displays a right arrow by the next line to be executed. The status window now indicates the line where execution halted, as well as information about the contents of the source window.

Figure 3



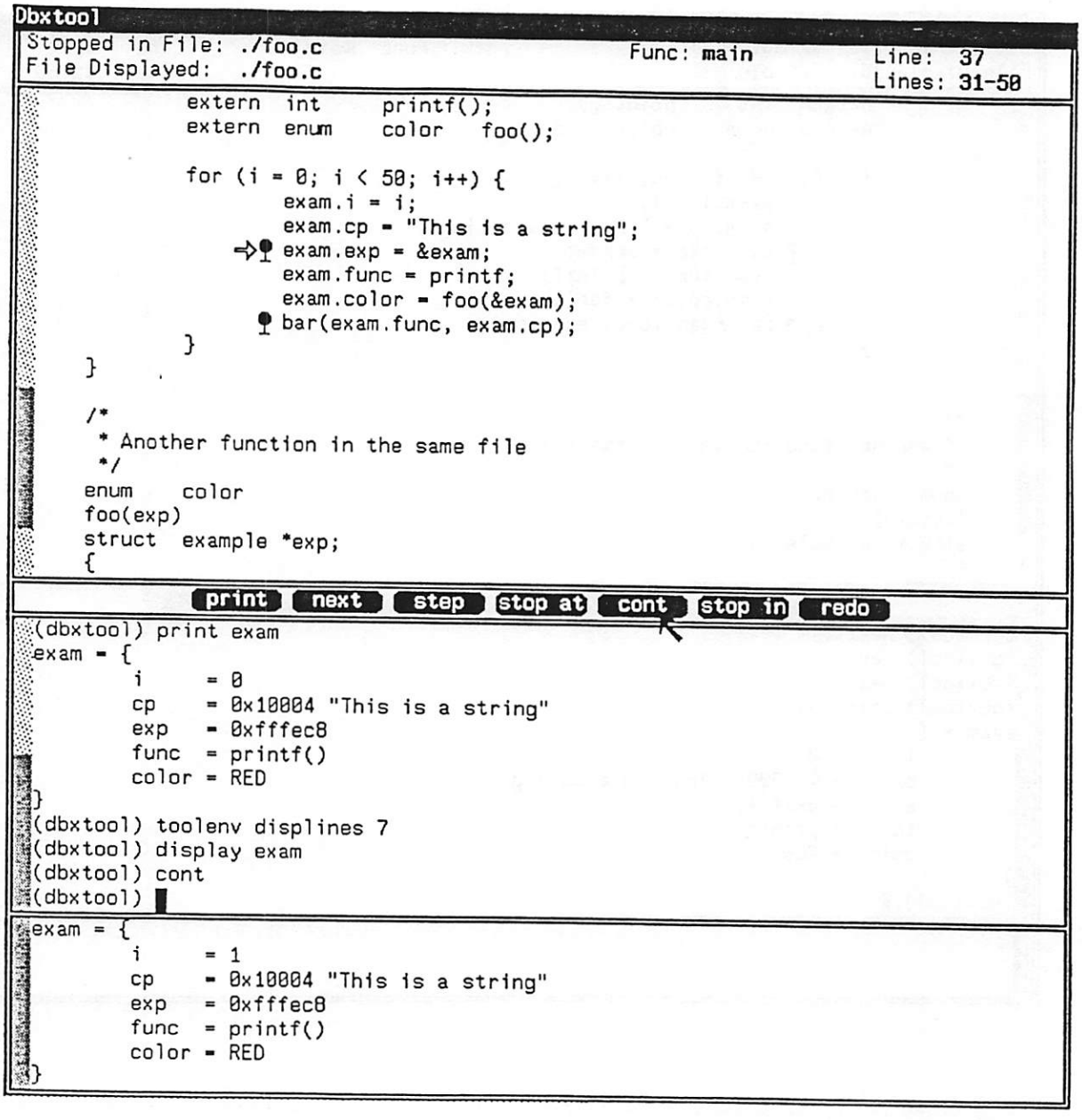
In Figure 4, the user has used the mouse to single step through three source lines (reflected in the three **next** commands shown at the top of the command window) and printed the value of the **exam** structure. The contents of the structure are displayed by printing each member in a format appropriate for its type on a separate line. Notice that the earlier commands and responses have scrolled off the top of the command window and the dark portion of the scroll bar is now reduced, indicating the portion of the command transcript currently visible in the window.

Figure 4



In Figure 5, the user has enlarged the size of the display window to seven lines (via the **toolenv displines 7** command) and requested that the contents of the **exam** structure be shown in the display window whenever execution halts (by the **display exam** command). The **cont** command then caused execution to resume until the next breakpoint was encountered. On the next trip through the **for** loop, the breakpoint at line 37 was encountered again and the contents of the **exam** structure were displayed in the display window. Notice that the value of **exam.i** in the display window is 1, while the old value still showing in the command window was 0.

Figure 5



5. Implementation

Dbxtool is implemented as two processes — a window-based front-end connected via a pipe to a slightly modified version of *dbx*. This division of labor has several advantages. *Dbxtool* invokes the standard *dbx* with a special command line flag telling it that it is running under *dbxtool*. Maintenance is reduced as there is only one version of *dbx* subject to enhancements and bug fixes. Therefore, users on ASCII terminals (who are restricted to *dbx* only) benefit from the same improvements as *dbxtool* users.

The user may wish to interact with *dbxtool* while the process being debugged is running, to move or expose the *dbxtool* window or to scroll a subwindow. If *dbxtool* were implemented as a single process, it would have to call the *wait* system call whenever the process being debugged is running and any interaction with it would be suspended until the next breakpoint were encountered.

Another advantage is the fact that *dbxtool* knows virtually nothing about the specifics of *dbx*. It could easily be adapted to a different debugger.

When *dbxtool* creates the child *dbx*, it creates a pipe between the two processes. *Dbxtool* assumes the responsibilities of displaying source code, indicating when a breakpoint has been encountered, and so on. When *dbx* reaches a point where it would normally perform one of these functions, it writes a small amount of information to the pipe leading back to *dbxtool*. *Dbxtool* reads from the pipe and performs actions such as positioning the source window, displaying or removing a stop sign, moving the right arrow to another source line, etc.

6. New Capabilities in *Dbx* and *Dbxtool*

We have added many new features to *dbx* and *dbxtool*, including faster initialization, source code searching, evaluation of all C operators except comma and the side-effect assignment operators, and have improved the stack walking facilities. The C operators include *sizeof* and type casts. Note that, in the following, references to *dbx* include *dbxtool* as well, unless it is specifically excluded.

6.1. Faster Initialization

To make full use of *dbx*, a source file must be compiled with the **-g** option, which causes the compilers to generate symbolic information for each variable, parameter, function, structure declaration, line number, etc. The assembler puts this information into the object file's symbol table. The loader takes a group of object files, relocates the appropriate *dbx* symbols, and passes ALL the *dbx* symbols on to the executable file.

In many cases, this approach creates a large amount of duplicated symbol information. Header files are the main culprit. They are frequently used to declare structures and are included in each source file that uses any of the structure definitions. Commonly, almost all the structure definitions in a program are declared in a single header file which is included in almost every source file.

For example, the Sun FORTRAN 77 compiler has one header file, **defs.h**, containing 26 structure and union definitions. The compiler source consists of 45 source files and 43 of them include **defs.h**. As a result, the symbolic information contained within **defs.h** would be included in the executable file 43 times. The executable thus would contain 1118 (= 43*26)

structure/union definitions, rather than 26.

To improve the speed of *dbx*'s initialization, we changed the representation of types in the *a.out* symbol table. The original 4.2 BSD *dbx* uses an integer to represent each type (its *type index*), and associates an index with each variable or function. The base types of C (**int**, **char**, **double**, etc.) are assigned the first twelve indices. When a new type is defined, it is assigned the next available index. Examples of programmer-defined types are **char ***, **struct foo { ... }**, and **int [10]**. The compiler assigns type indices linearly through a source file. The base types are first, followed by the programmer-defined types, regardless of whether a new type comes from a header file or the source file. Therefore, the type index of a particular type depends upon the order in which the header files are included.

We now represent types by ordered pairs consisting of a *file index* and a *type index*. The file index determines which file defined the type and the type index uniquely identifies the type within the defining file. Therefore, changing the order in which header files are included does not change the type indices of the types defined within a given header file. If a header file defines ten new types, they will always have type indices one through ten.

We added two new *a.out* symbol table types (N_BINCL and N_EINCL) that delimit a group of symbols defined by a header file. The loader builds a table of the symbols generated by each header file and, if two instances of the same header file produce the same symbolic information, only one copy of the symbols is written to the executable file. The redundant copy of the symbols is replaced by a third new symbol table type (N_EXCL) indicating that a group of symbols has been excluded.

During initialization, *dbx* remembers the type information associated with each header file. When *dbx* reads an N_EXCL symbol, it finds the remembered header file entry and maps the header file's type information into the pool of active types. Future references to types defined in the header file associated with the N_EXCL symbol can now be resolved.

When this approach is applied to the Sun FORTRAN 77 compiler, it reduces the size of the executable by 41% (from 931K bytes to 553K bytes), the number of symbol entries in the executable by 29%, and the real time spent in initializing the debugger by 80%.

6.2. C Operators

We enhanced *dbx* to support all C operators except the comma operator and the side-effect assigment operators. We added the following operators:

(type)	type casting
sizeof	sizeof an object
<<	left shift
>>	right shift
&	bitwise AND
|	bitwise OR
^	bitwise XOR
~	complement
!	not
? :	conditional expression

Type casting has proven to be the most popular addition. Some modules use the **char *** type as an opaque pointer (a pointer whose actual type is known only to its implementing module, not to its clients). The UNIX kernel and the Sun window system are examples of such modules. Type casting in *dbx* allows the user to examine what the pointer points to under an appropriate type interpretation.

6.3. New Commands

We have added several new commands to *dbx*. The most useful ones walk the stack and provide string searching in the source code. The 4.2 BSD *dbx* provided the **func** command to walk the stack. It was inadequate for recursive routines and did not provide an easy way to change the context to the calling routine. The **up** and **down** commands implement stack walking relative to the current routine.

When a breakpoint is encountered, the routine containing the breakpoint becomes the current routine. The **up** command moves the notion of the current routine up the call stack (towards **main**), and the **down** command moves it down the stack (towards the current stopping point). In *dbxtool*, the source window is scrolled to include the line containing the call and it is highlighted for one second. The **up** and **down** commands in *dbx* were originally implemented by Mark Linton at Stanford University.

Regular expression string searching is provided by the / and ? commands, via the *re_comp* and *re_exec* library routines. The / command searches the current source file from the current location forward and the ? command searches from the current location backward. In *dbxtool*, the source window is scrolled so as to include the line containing the matching expression and it is highlighted for one second.

We also added the **clear**, **display** and **undisplay** commands. **Clear** provides the inverse of the **stop at** and **stop in** commands, i.e. it removes breakpoints. Within *dbxtool*, one can remove a breakpoint by selecting the line containing it and clicking on the **clear** button.

The **display** command takes a list of expressions to be shown in the display window whenever execution halts. The **undisplay** command removes expressions from the set shown in the display window. In the line-oriented *dbx*, the **display** command functions as a set of automatic **print** commands performed whenever execution halts.

7. More Exotic Debugging Facilities

We have enhanced *dbx* and *dbxtool* to provide debugging of arbitrary (non-child) processes, multiple-process programs, and the UNIX kernel.

7.1. Arbitrary-Process Debugging

The 4.2 BSD *dbx* supports debugging of live processes and post-mortem debugging. To debug a live process, it must be a child of *dbx*. We removed this restriction.

For example, if the **sendmail** daemon is behaving strangely, one can attach *dbx* to it and begin debugging it with **sendmail** already in an odd state. If the problem can be corrected (possibly by patching a variable), the daemon can be detached from the debugger and allowed to continue running.

7.2. Multiple-Process Debugging

Multiple-process debugging occurs when more than one process must be debugged simultaneously. Some examples are network programs consisting of a server and a client, daemons that wait for an event or resource and then spawn a child to do some work, and the *dbx* and *dbxtool* components of the debugger.

There are two approaches to multiple-process debugging. Either a single debugger can know about more than one process and provide access to them all, or there can be an instance of the debugger for each process being debugged. We chose the latter. The first solution poses two major problems, process resolution and interruption. Process resolution is the problem of identifying for which process a given debugger command is intended. Commands such as **print i**, or **stop at 147** can apply equally well to any process being debugged. The notion of an *active* process (the object of debugger commands) and the ability to switch between processes are essential.

Interruption is the problem of one process being stopped when another encounters a breakpoint. The user may be issuing debugger commands to examine the state of the stopped process when an active process encounters a breakpoint. In *dbxtool*, the source code surrounding a breakpoint is displayed in the source window whenever a breakpoint is encountered. This creates confusion in the source window — which source should be displayed? If the proper source code is not currently displayed, the user must at least be notified that a process has stopped. When a process stops, should it become the active process?

In a window environment, multiple instances of the debugger provide a more elegant and effective form of multiple-process debugging. The user does not have to push, pop and query the status of the processes being debugged. He can easily switch between them by switching between instances of the debugger.

7.3. Kernel Debugging

We added the ability to debug the UNIX kernel to *dbx*. When debugging the kernel, *dbx* uses the page maps within the kernel (or core image) to map addresses. The **proc** command allows the user to specify which process's user structure is mapped into the kernel's *uarea*. Therefore, the user can inspect the kernel stack trace of any process that is active.

7.4. Implementation

To implement arbitrary-process debugging, we added the ATTACH and DETACH requests to the *ptrace* system call and added a field to the *proc* structure. The ATTACH request checks permissions, stops the given process (by sending it a SIGSTOP) and returns a 0177 status (''process is stopped'') to a subsequent *wait* system call. The DETACH request frees the process from the debugger. We added a field to the *proc* structure containing the process id of the tracing (debugger) process so that it is no longer required to be the debuggee's parent.

We also added six other *ptrace* requests — read/write the registers (GETREGS, SETREGS), read/write the text segment (READTEXT, WRITETEXT), and read/write the data segment (READDATA, WRITEDATA). The GETREGS and SETREGS requests significantly reduce the number of *ptrace* calls necessary for single-stepping operations. The READDATA request has greatly reduced the number necessary to print the call stack. Overall, the number of *ptrace* calls has been reduced by about two-thirds.

For both user-process and kernel debugging, we have tried to make *dbx* immune to changes in the the kernel's internal data structures. Most UNIX debuggers require that the debugger know the format of the *user* structure to read and write a process's registers. When the offset of the registers within the *user* structure changes, the debugger must be recompiled. The GETREGS and SETREGS requests isolate the debugger from changes to the *user* structure as they define a position-independent interface for retrieving and setting a process's registers.

For post-mortem debugging, UNIX systems have traditionally written a copy of the *user* structure to the beginning of the core file. The debugger again uses its knowledge of the *user* structure to access the process's registers. Instead, we have defined a core file interface to provide access to information of interest to debuggers, such as the registers, the *a.out* header, the killing signal number, the command name, etc. This interface will remain unchanged as modifications to the *user* structure occur.

For kernel debugging, we have added several new symbols to free *dbx* from having to know kernel internals. The symbols tell *dbx* the size of a *proc* structure, the offset to the *p_pid* and *p_flag* members of the *proc* structure, the number of pages in the *uarea*, the offset to the registers in the *uarea*, etc. These symbols allow *dbx* to find a process's registers and to search the *proc* table even across changes in the formats of the *proc* and *user* structures.

8. Future Improvements

There are many possible future directions for work on *dbx* and *dbxtool*. The following describes some of the possibilities under consideration.

8.1. Symbol Table Reorganization

We achieved a significant improvement in the speed of *dbx*'s initialization by reducing redundant symbol table information; however, an entirely new approach is desirable. Currently, *dbx* reads all the symbols before beginning a debugging session. Thus initialization time is roughly proportional to program size. In the course of a single debugging session, on the other hand, only a very small percentage of the symbol information is ever actually used. The symbol table should be reorganized so that information can be retrieved quickly as it is needed.

8.2. Multiple Language Support

Dbx was intended to be a multiple-language debugger. There are two basic approaches to multiple-language debugging — the debugger can define an input syntax of its own and require users to use its input syntax independent of the source language being debugged, or the debugger can support the input syntax of each of its supported source languages. The current implementation of *dbx* chose the first approach. *Dbx*'s input syntax is a subset of C with a few Pascal-oriented extensions and one concession to FORTRAN 77.

In the long run, we believe a multiple-language debugger must support the full syntax of each of its supported languages. If the user can select an expression with the mouse, the debugger should be able to evaluate it. This requires the debugger to contain a scanner, parser, and evaluator for each supported language. Depending upon the range of supported languages, the scoping rules within the symbol table may be affected as well.

Conceivably, these major functions could be table-driven. However, the problem becomes more involved when multi-language programs are considered. If, for example, a program

consists of both C and Pascal code, then both the C and Pascal syntaxes and evaluation mechanisms must be present. When execution has stopped in a C routine, the debugger should expect C expressions, and when it is stopped in a Pascal routine, it should expect Pascal expressions.

8.3. Remote Debugging

As described above, *dbxtool* is implemented as two processes — the window interface process (*dbxtool*) and the debugger (*dbx*), which communicate through a pipe. They send information about the locations of breakpoints, breakpoints that have been encountered, which lines to display in the source window, etc. through the pipe.

If the pipe interface were replaced by a remote procedure call (RPC) interface, then *dbx* and the process being debugged could be run on one system on the local area network, while *dbxtool* and the user could be on another. Thus, *dbxtool* could be used to debug processes on machines without Sun consoles and on other workstations over the network.

This opens the door to debugging processes on machines with different architectures. For example, by placing the appropriate RPC calls in the VAX version of *dbx*, one could use the power of *dbxtool* on a Sun Workstation to debug programs running on VAXes. The RPC calls define an abstract remote debugging interface. The debugger on the remote system would not even have to be *dbx* — it could be an entirely different debugger (and the remote system would not have to UNIX either!), it need only make the appropriate RPC calls.

8.4. The C Preprocessor and the Peephole Optimizer

Dbx should know about C preprocessor #define names and macros. The goal of selecting any expression in the source window and evaluating it cannot be realized until preprocessor names are supported.

Currently, the **-g** (debugging) and **-O** (optimizer) compiler options are mutually exclusive. This is overly restrictive. Thoroughly combining debugging and optimization is a topic requiring further research; however, given the state of peephole optimization in the UNIX environment, some pragmatic options are available. The only debugging information that the peephole optimizer invalidates is the line number information. The optimizer does not assign variables to registers, expand procedures inline, or do anything else that would severely impact debugging.

8.5. Integrated Environment

Dbxtool provides a source window for examining source code and selecting variables or expressions. A more integrated environment would allow editting in the source window and would automatically recompile modified files. The user would then have one tool providing the editting, compiling, and debugging functions.

9. Conclusions

Dbxtool has been an overwhelming success. Its users uniformly view it and the new capabilities in *dbx* as major improvements over the previous versions of *dbx*. A user with extensive experience with the Mesa debugger Pilot at Xerox PARC has commented that *dbxtool* provides somewhat less functionality, but a better user interface.

Several lessons were learned in the process of designing and implementing *dbxtool*:

- The expenditure of a large fraction of two engineers' time for two months on the design of the user interface, with several presentations to interested parties and much feedback, was well worthwhile. The resulting design was, we are convinced, much superior to the initial one.
- Changing the style of interaction with the debugger has lead to needs and desires for additional capabilities. In particular, the **button**, **display**, **undisplay**, and **toolenv** commands (which control various aspects of the *dbxtool* window) became necessary. String searching became highly desirable and so was added to the source window. More recently it has been noticed that it is also desirable to have in the command and display windows.
- The choice of having multiple selection interpretations and postfix command buttons, while different from the approach taken in *dbx*, is the correct one for the mouse-based version. It allows flexibility and minimal effort in command construction.
- The separation of the debugger and the user interface into distinct processes with a narrow communication channel between them is very valuable. It allows work on the debugger to be reflected in both the window and non-window versions and will simplify the introduction of a remote debugging capability.
- Our experience has demonstrated ever more strongly that it is important for the compilers and the debugger to speak the same language. The lack of this we consider to be the greatest failing in our current debugger. Not only do the C syntax and evaluation rules differ somewhat from those that should be supported for Pascal and FORTRAN 77, but the evaluation model in the debugger currently differs slightly from that in the Sun C compiler.

Acknowledgements

We thank Wendy DuBois for her early and significant contributions to the design and implementation of *dbxtool*. We also thank the members of the Programming Languages and Programming Environments Groups at Sun for their contributions to its design, Bill Shannon for his modifications to the Sun UNIX kernel to support the arbitrary-process debugging and kernel interface improvements, and Mark Linton for his continuing support and for the improvements he has provided from his version of *dbx*.

The Cloned Tree Method of Revision Control
or
A Rich Person's Revision Control System
or
How I adapted the UNIX file system and tools that manipulate it to perform project revision control

David Yost

Consultant
8464 Kirkwood Drive
Los Angeles, CA 90046
213/650-1089

ABSTRACT

When you work on a large software project, your files are spread out in a directory tree. Over the years, I have feathered my tool nest with new tools and modifications to standard UNIX tools to manipulate these linked trees easily. Along the way, I had a brief bout with SCCS, and I liked the idea, but ultimately went back to improving and using the tree tools. I simply keep each old version as a read-only directory tree of files (linked where identical, to save disk space). Keeping all old versions of files on-line in this way takes more disk space than SCCS or RCS (hence Rich Person's Revision Control System), but versions are easily and quickly accessible, and you can snapshot an entire consistent version easily and name it with the date and, if you like, some number like 1.3.2.1.

A large software project comprises program source files, documentation files, linkable binary files, executable binary files, revision notes, makefiles, shell scripts, etc. Typically, these files are organized into a directory tree structure. To see a snapshot of your project, you simply look at all the files in this tree. But what if several people are working on the project, and you are making releases periodically? How do you keep track of the different versions? How do you integrate the work of all the contributors? Well, you might use Revision Control tool sets designed to help you. Two major well-known tools have been available on UNIX† for some time, The Source Code Control System from AT&T and RCS from Purdue University. These are collections of programs that maintain revisions for you. They keep one master archival file for each file in your project, from which you can extract any past version of the file with revision information embedded in it. They also provide a place to note the changes made at each step in the chain of revisions, administrative control over who is working on what, etc.

Another way to deal with the problem of the moving target of a large project is to simply take snapshots of the whole directory tree every time you want to freeze a revision, i.e. make a copy of the whole tree. This takes more disk space (hence the Rich Person's Revision Control System), but it has the advantages of simplicity, convenience, and speed. At any time, you can

† UNIX is a trademark of AT&T Bell Laboratories.

refer to a past version of any file or group of files without needing a special program to extract files so you can compile them, edit them, or give them to someone. The disk space required can be cut down, too, by linking files which are the same.

The Cloned Tree Method (CTM)

This 'system' of revision control evolved over a number of years before acquiring a name. Finally in 1982, when we were using it to keep track of kernel development at Fortune Systems, one of my colleagues came up with the phrase cloning a tree to describe the operation of making a linked copy of the master source tree. Thus, The Cloned Tree Method was born.

Cloning a Tree

In the late seventies, there was an old Version 6 **copy** command written at the Rand Corporation by Keith Davis. This command had a **—r** option which made it work recursively on a directory tree, and a **—l** option which made it link if possible instead of copying. After Version 7 came on the scene, I rewrote **copy** at Rand in 1981, and I have added more options to it since then, as they were needed for CTM. I'll refer to this new version as **cpt** (copy tree) to avoid confusion with standard **cp**.* The ability to do recursive linking is at the very heart of CTM.

Everyone working on a project should work on the same file system so files can be linked. This being the case, you clone a tree by saying

mkdir *ToDir*
cpt —rl *FromDir/. ToDir*

Since snapshotting an entire consistent version is very easy, it is a good idea to take a snapshot every time you get to a new plateau, even if it isn't time to make a release.

Links

To save disk space, all the files in a newly-cloned tree are linked to their counterparts in the original tree. Therefore, before changing a file in your tree, you must break its links to other trees by renaming or deleting the file in your tree and writing out a new file. To ensure that this policy is maintained, all files must be read-only, and no one must work as the super-user. A simple

chmod a—w `find . —print`

should do the trick. However, if your project has too many files in it, this command may fail because the argument list to the **chmod** command will be too long. In that case, you can use the **—r** (recursive) argument added to **chmod** to deal with this.*

Making your files unwritable guards against disaster, but you still have a problem on your hands: most UNIX tools don't know that you want them to unlink* a file before writing to it, and they will try to write to these unwritable files. The four most common ways to clobber linked files are:

Fortune Systems did in fact adopt this version as their standard **cp** command.

This limit on the number of characters that can be passed in an argument list could and should be removed from the UNIX kernel. It is a constant source of irritation, and encourages the addition of a lot of clutter to tools in order to work around it, such as the **—r** option to **chmod**.

In the UNIX file system, there can be several file names referring to the same file data. These are all said to be 'links' to the file. The term 'unlink' is the UNIX term for deleting a file name from a directory. If the name deleted was the only link to the file, the unlink operation deletes the file data from the disk; otherwise, the data and the other links to it remain. I will often use the term 'unlink' instead of 'delete' or 'remove' where it seems more appropriate.

1. *Editor updating*

The Rand Editor and all its derivatives have always updated files by renaming the original file to a backup name and creating a new file. This is just what you want for CTM, and in fact, this feature of the Rand Editor was an early inspiration for CTM. If the file 'notes' is updated by the Rand Editor, the original 'notes' file is renamed to ',notes' and a new 'notes' file is created. At Fortune Systems we added this leading-comma backup file convention to **vi** ('non-unlinking', also known as 'in-place', update is still available as an option, as in the Rand Editor).

2. *Shell output redirection* (**>**)

At Fortune Systems we added a C-shell built-in variable, **unclobber**, similar to the already existing **noclobber** variable. When **unclobber** is set, an output redirect moves an existing output file 'out' to ',out' and writes out a new file (you can force overwriting with '**>!**'). So far, no similar mods have been made to **/bin/sh**.

3. *The* **cp** *command*

The UNIX **cp** command is not of much use with CTM because it always overwrites existing files. The **cpt** command will unlink the destination before copying if you run it with the **—u** (unlink) option or with the **—l** (link if possible) option. I would have made **—u** the default, but then **cpt** would not be compatible with **cp**.

4. *Commands run from the* **make** *command*

To keep **make** from overwriting files, every rule must be careful to unlink files before writing to them. Usually this means inserting a command like this into each rule:

—rm —f $@

This also means that you have to include explicit rules for everything, including such things as '.c.o'. The **make** command should be modified to take an option that forces deleting objects before making them.

Master Directory Names

Designate a master directory for the project. In this directory, keep the roots of all the trees containing all the frozen versions, and possibly one or more writable complete development trees in which it is OK to make changes. Name the root of each of frozen tree with the date of the most recently modified file in the tree, as in '850508' for May 8, 1985. Further identification of a version can be added after the date, as in '850508—1.7'. If you want to freeze a version taken from some other machine or person, you might have a name like '850508—1.7—chris'. This naming method has the advantage that you can do an **ls** in the master directory and see all the frozen trees in chronological order. (It was upon the discovery of this refinement that CTM advanced from an *ad hoc* system to a more mature system that required a name.)

What's in the Frozen Trees

You can keep your frozen trees 'clean' or you can leave the **make** results (executable binaries, compiled libraries, nroff'ed man pages, etc.) in the tree. If you keep a frozen tree 'clean', and at some time you want to do a **make** there, just clone it to a working tree with writable directories, and do the **make**.

One of the nice features of CTM is that your working tree can have lots of test files, hacked versions, note files, and junk in it, but the frozen trees contain only the real sources. In the root of each tree, I keep a file called 'Files' which contains a complete list of the real files in the tree, Nothing which is generated from the other files of the tree with **make**, etc. is in this list. You can use this list to save only the real files from your work tree with the **cpt —p** option which copies each file into its corresponding position in the destination tree.

cd *WorkDir*

```
cpt —lp `cat Files` ../850508
```

Another way to do this is to clone a recent frozen tree into the new tree, then do a **cpt —lrx** from your work area to the new tree, then freeze it. The **—x** option to **cpt** copies files only if the destination exists. This option is also very useful for cloning subsets of a tree: first you make a new tree that has only the files you want in it, then you do a **cpt —lrx** from the source tree to the new tree.

User and Group Ownership

The user and group ownership of the files in the frozen directories can be useful information. You should change the group ownership on the files in the frozen trees to a special group, such as 'frozen', so that if you are looking through the files in some work directory, in which most of the files are links to frozen files, you can identify the frozen files easily by their group owner. You might want to go so far as to change the ownership of the frozen files, too. On a 4.2bsd system, for instance, it can be very handy to change the ownership on all the original files that came with the distribution to a userid named 'bsd4—2'. If you also change all these to be unwritable, and are careful when you are acting as root so as to never clobber the contents of one of these files, you can always tell at a glance which files are vanilla and which are not. A new tool, **chowngrp**, which is a superset of both the **chown** and **chgrp** commands is useful. It takes a **—r** (recursive) argument and will change owner, group or both.

A few years ago, I used CTM to organize a set of trees of several releases of UNIX: Version 6, Version 7, Version 7 Update, Phototypesetter Version 7, UNIX/32V, 4.1bsd, and System 3. I wanted to see which sources had changed and when. I extracted a tar tape of Version 6, froze it (made it unwritable), and made all the files owned by a user called 'unixv6' and a group called 'origtape'. Next, I extracted a tar tape of Version 7, froze it, and made all the files in its tree owned by 'unixv7' and the group 'origtape'. Next, I ran a modified version of Berkeley **diff** to which I had added a **—q** option to be quiet about ASCII file differences and a **—L** option to link from tree1 to tree2 when the files were identical. This **diff** linked all identical files from the Version 6 tree to the Version 7 tree. The result was that I could look in the Version 7 tree and see at a glance by the ownerships which files had not changed since Version 6. I continued along these lines on to the later versions. It got messy at the point when Berkeley started putting SCCS id strings into files without making any other changes. At that point, I made a specially hacked version of this **diff** that would ignore source lines containing SCCS id's. The resulting revision history database was very useful in maintaining UNIX code.

Another useful side effect of always modifying files by unlinking first and then writing out a new version is that you can see who last modified a file by looking at its owner.

Modify Times

When you look through a tree to see what's what, the modify time on the files can be very useful information. Since I have been using CTM, I have developed a new sense of the meaning of the modify time on a file. To me it means the last time the bytes in a file were changed relative to each other. It's as if the data in the file has its own existence, and wherever it is moved to, it should have the same modify time unless the bytes are rearranged in some way. To avoid clobbering file modify times, you should use **cpt** with the **—t** option, which sets the modify time of the destination to that of the source. Or, if you are moving a file over a network or to and from tape, you can use **tar**, which will preserve modify times.

Change Notes

When you are going to freeze a new version of a tree, you should do a **diff —r** between the last version tree and the current one, and write up one or more files describing the changes and put those files in the new tree. This has to be done by someone who is familiar with all the changes. This is in contrast with SCCS and RCS which ask you what you did each time you

change each file. This can lead to either very cryptic or very overdone change notes, which are spread out all over the place and can be almost useless under the shear mass of detail. Consider the case where what is really one simple change affects several files. Should you repeat the same comment several times, once for each affected file? And what of the changes which didn't work and were redone later after other changes had been made, so the comments couldn't really be backed out?

Change notes written at the time of freezing the complete new version can give a consistent, concise view of all the changes in the whole system, including what files were affected by each change, and in what way they were affected.

Integrating Changes

Often, several people are working on different parts of the code at once. This is not bad if each person who makes changes can always point to the frozen tree from which they cloned their tree. Before making changes, they clone their own tree from a frozen tree. Next, they make whatever changes they want. If the tree they cloned from is still the latest integrated version, it's easy easy to integrate these changes into the master version. If not, then the integration must be done by someone intimate with the project who will understand how changes made by different people might interact. At Fortune, we called the person who did this the 'Grand Integrator'. This Office bounced back and forth between Rick Kiessig, the head of the kernel group, and me.

When it is time to integrate software, you have a Starting Tree and trees descended from it: the Changed Tree, and the Current Tree. Your job as Grand Integrator is to integrate the Changed Tree into the Current Tree to end up with a Final Tree. First you clone the Current Tree to serve as a first cut at the Final Tree. Next, you look at a **diff —r** between the Start Tree and the Changed Tree. Next, you look at a **diff —r** between the Start Tree and the Current Tree.

Next, you work on each file that is changed in the Changed Tree. For this step, there are some useful modifications to the **diff** command. The **—j** option causes **diff** to print all common lines preceded by a '¦' character, in addition to the changed lines which are preceded by the usual '<' and '>' characters. The **—t** option causes **diff** to insert a tab character (instead of the usual space character) between the '<', '>', or '¦' change indicator characters and the beginning of text from the compared files. Thus, you can use **diff —jbt** to make a combined file which you can edit and distill down to an integrated version.

Another tool which I personally cannot do without at this step is the Rand Editor (actually my own derivative version of it). It lets you work on two files at once, the 'current file' and the 'alternate file', and you can quickly flip back and forth between them on the screen and see the differences. This feature was given the name 'video diff' by Michael O'Brien.

Case 0: Current Tree file has no changes. Changed Tree file has no changes.

Do nothing.

Case 1: Current Tree file has changes. Changed Tree file has no changes.

Do nothing.

Case 2: Current Tree file has no changes. Changed Tree file has changes.

Install the Changed Tree file in the Final Tree with **cpt —l**.

Case 3: Current Tree file has more changes than Changed Tree file.

Do a **diff —jbt** between Start file and Changed file. Edit the Final file, and install changes from the **diff** output file.

Case 4: Current Tree file has fewer changes than Changed Tree file.

Do a **diff —jbt** between Start file and Final (same as Current) file. Edit a copy of the Changed file, and install changes from Final file. Move copy of Changed file to Final file.

Case 5: Current Tree file has same amount of changes as Changed Tree file.

Do a **diff —jbt** between Start file and Changed file. Do a **diff —jbt** between Start file and Final (same as Current) file. Flip back and forth between them and reduce the Final file to the integration of the two.

Note that cases 3, 4, and 5 above can be very difficult, but the tools outlined here help a lot.

Saving Trees to Tape

Eventually, you will want to save disk space by deleting obsolete or intermediate versions. You can use **du** to find out how many blocks are used in a tree, but this is not the same thing as the number of blocks that will be released if you delete the tree. I modified **du** to tell you this information.

If you make a **tar** tape of a frozen version, it can be recalled later without using a ridiculous amount of disk space by using the new **X** option to **tar** like this:

cd *ExtractDir*
tar X *../FrozenDir*

For each file on the tape, **tar** will extract the file as a temp name in the directory where it belongs and compare it with the corresponding old file in the *FrozenDir* tree. If the files are identical, **tar** will delete the temp file and link from the old file to the new file. If not, it will move the temp file to the new file.

Summary of the Tools

Here is a summary of the new tools and modifications to existing tools which are useful with CTM. I will attempt to gather together all of these tools and modifications and lobby for their inclusion in 4.xbsd or at least in the Usenix distribution.

The **cpt** command

A **—r** option to work recursively on a directory tree.

A **—l** option to unlink the destination and then link the source to the destination.

A **—t** option to set the modify time of the destination to that of the source.

A **—x** option to copy only to files already existing in the destination.

A **—p** option to copy each file into its corresponding position in the destination tree.

And many, many more options.

The **chowngrp** *command*

Changes owner and/or group, and takes a **—r** option to work recursively on a directory tree.

The Rand Editor

Perfect for CTM. Updates by moving file to ',file' and writing out a new file. 'Video diff' feature aids in integration.

Modifications to **vi**

If an environment variable is set, update by moving file to ',file' and writing out a new 'file'. Override this behavior with **w!**.

Modifications to **diff**

With the —**L** argument, if a file is identical in tree1 and tree2 will be unlinked in tree2 and linked from tree1 to tree2.

With —**j** option, includes in the output all identical lines, prepended with '¦'.

With —**d** option, directory compares also examine files beginning with '.'.

With —**t** option, the character after the '<', '>', or '¦' is a tab instead of a space.

With —**q** option, diff simply reports when ASCII files differ without going into detail.

Modified to work in the **diff file ... dir** case.

Modifications to **csh**

If **unclobber** is set, **> out** redirection is accomplished by moving the previous version 'out' to ',out' and writing out a new file.

Modifications to **tar**

The **X** option to **tar** allows extracting only files that differ from those in a specified tree, and linking the rest from that tree.

The **C** option to **tar** allows comparing files on tape against the tree rooted in the current directory.

The **U** option when extracting causes files to be unlinked before they are created.

And several more fixes and changes.

Modifications to **chmod**

The —**r** option causes it to work recursively on a directory tree.

Modifications to **du**

The summary of total blocks now also says of which *nnn* are not linked elsewhere.

The **du** command used to have a fixed link table, which would often be overrun when running **du** on large, heavily-linked trees. I modified it to allocate memory for the link table so it can handle as many links as memory size will permit.

Other Considerations

The symbolic links introduced in 4.2bsd are not appropriate for CTM because if you delete or change the file pointed to by a symbolic link, your data is lost or changed. Hard links to unwritable files have just the right properties for CTM.

A drawback of SCCS and RCS is that they can't maintain old versions of binaries. It should be obvious that CTM has no problem with that.

SCCS introduced the concept of compiling identifying strings into binaries and provides the **what** command to print out those strings. This is a good feature which can be used even if you don't use SCCS. You keep the version information in a file, usually in the root of the tree, and you set up your makefiles to automatically create a version.o file from this version information and to link it in with all binaries.

Converting from SCCS or RCS to CTM is not very hard. You just make your version trees and extract the versions into the appropriate trees and run **diff —rqL** successively from the oldest tree to the newest. If you have files which differ only in their SCCS id strings, you may have some hand work to do.

Future work

The **mv** command doesn't preserve modify times. It should.

Other shells besides the C-shell need to be taught not to clobber files.

Acknowledgements

The addition of the **—r** option to **diff** by Bill Joy while he was at Berkeley was a major step in the development of CTM. Mark Stein at Fortune Systems modified **vi** and **csh** to avoid overwriting files. John Lowry at Fortune Systems modified **diff** to add the **—j, —b, —d, —t,** and **diff file ... dir** features.

Manual Pages

Accompanying this article, you should find manual pages for **cpt, diff,** and **tar.**

NAME

cpt — copy or link files and directory trees

SYNOPSIS

cpt [—abdilnNoprsStTuvVx] [—] file1 file2

cpt [—abdilnNoprsStTuvVx] [—] file ... directory

DESCRIPTION

Cpt copies *file1* to *file2*. In the second form, one or more *files* are copied into *directory*, which must exist, and the last element of the file path name is used as the name of the new file in *directory*. Links between source files will be preserved in the destination wherever possible. *Cpt* can be used to copy whole directory trees with the **—r** (recursive) option described below.

If there is a single source argument and it is a special file, it is opened and read as if it were a regular file. In the multiple-source-argument or recursive case, special files are ignored.

If a destination file exists, its owner and mode are unchanged, and the source is copied onto it. Otherwise it will be created with the ownership of the invoker and the mode of the source, and the setuid and setgid bits will each be set only if the new file is owned by the same user id and group id as the source, respectively. If the source is not a regular file and the destination is new, then *cpt* creates the new file with permissions rw—rw—rw— subject to the umask.

Many of the features of *cpt* listed above can be modified with one or more of the following options, which may be clumped together as in '—rv' or separated as in '—r —v':

—l Link instead of copying if source and destination are on the same file system. If a link can be made and the destination already exists, *cpt* will unlink it before making the new link. Note that this is different from the behavior of *ln*(1). If you want a link to fail when the destination exists, as it would if you were using *ln*(1), then use **—n** along with **—l**.

—r Recursive. Copy each source directory and its entire subtree. Make new subdirectories in the destination directory as necessary. Without this option, directories are skipped.

—p Use full path names. Form each destination name by appending the entire source path to the destination path. This is especially useful for copying selected files from one tree to another. Example:

```
cpt —p d1/d2/f1 x
```

will make x/d1 and x/d1/d2 if necessary and copy d1/d2/f1 to x/d1/d2/f1.

—i Prompt the user with the name of the file whenever the copy will cause an existing file to be overwritten. An answer of 'y' or 'Y' will cause *cpt* to continue. Any other answer will prevent it from overwriting the file.

—a Ask on all files before copying. If **—r** is selected, sets **—d**. Answer is interpreted as in **—i**.

—d Ask on all but regular files before copying. If *cpt* is copying recursively, and you decline to examine a directory, that directory and recursively all its contents are skipped. Answer is interpreted as in **—i**.

—n Copy a file only when the destination does not already exist.

—x Copy a file only when the destination already exists. Do not make any directories.

—b Copy a file only when the destination doesn't already exist or its inode modification time (st_ctime) is older than the source.

—s Do special files. If **—l** is not set, or it is set and *cpt* can't link, and you are super-user, do a *mknod*(2) so that the destination file is a device just like the source.

—u If the destination is a regular file and already exists, unlink it before copying. This is

useful when the destination file is multiply-linked and you don't want to affect all the links. It is also useful when you want all the destination files to be owned by you and to have the same mode as the source.

—o Set the group and user id ownership of the destination to that of the source. Only super-user can do this.

—t Set the file modification and access times of the destination to that of the source. Only works if you are super-user or you own the destination file. To be sure that you will own destination files, use the **—u** option.

—T Same as **—t** plus the access time of the source is unchanged by the copy.

—N No copy operation. This allows you to do a **—o** and/or **—t**, to destination file(s) which already exist. No file copies take place, and no directories are made when this option is in effect. Can not be used with **—l**, **—n**, or **—u**.

—v Verbose, partial. Print a line on the standard output noting each copy operation which is skipped because of conditions specified by the **—n** or **—b** options.

—V Verbose, full. Print a line on the standard output noting each copy operation, whether it is to be done or skipped, each source directory that is examined, and each destination directory that is created.

—S Replace an argument of "=" with a list of newline-separated files from the standard input. If this option is in effect, it is assumed that you are copying multiple files to a directory even if there is only one file in the list from the standard input, and the destination argument must be on the command line.

— The null option '—' indicates that all the arguments following it are to be treated as file names. This allows *cpt* to work with files whose names start with a minus.

Cpt attempts to read 16K bytes at a time, and writes out the same amount it reads each time. Thus it is usable with tapes or other devices that have large or irregular block sizes.

Cpt will not attempt to copy a file onto itself.

Here are some examples. Assume the following directories and files:

```
d1/f1, d1/d2/f2, and d3
```

where d1, etc. are directories, and f1, etc. are files. Then

```
cpt d1/f1 d3
```

would make d3/f1, and

```
cpt —r d1 d3
```

would make d3/d1/f1 and d3/d1/d2/f2, and

```
cpt —r d1/. d3
```

would make d3/f1 and d3/d2/f2.

You can effect a recursive move across file systems by doing a 'cpt —r' followed by a 'rm —r'. The *mv*(1) command is best for moving a subtree within a file system.

SEE ALSO

cat(1), ln(1), mv(1), rm(1)

DIAGNOSTICS

Copy is normally silent about its operation. If an error occurs, a message will be printed to the standard error output. If a file is not copied for some other reason, this will be noted on the standard output unless this was because it did not meet a condition specified by **—n** or **—b**. For more verbosity, use **—v** or **—V**.

Cpt does not stop prematurely on errors. Exit status is '0' if all files are copied OK. If any file or directory was skipped for some reason, the exit status is '1' unless any file or directory was supposed to be copied, but couldn't because of an error, in which case the exit status is '2'.

Errors may result in partially-written destination files.

AUTHOR

Dave Yost, (original version for The Rand Corporation)

BUGS

In the recursive case, should check that the destination is not a subtree of the source.
The ownership, and times of directories created under the **—p** option cannot be set to match the source directories, and in the mode is set to 'rwxrwxrwx'.
Links may be 'preserved' in the destination to a file which did not copy successfully.

MORE WORK

Provide a **—m** option to move the files, i.e. delete the source files and directories after the copy.
Provide an option to sort the filenames in each directory alphabetically before copying, and copy directories first, then files.
Provide a way to copy to multiple destinations at once.

NAME

diff — differential file and directory comparator

SYNOPSIS

diff [—b] [—cefhj] [—l] [—L] [—rfR] [-q] [—s] [—Sstring] [—t] *dir1 dir2*
diff [—b] [—cefhj] *file1 file2*
diff [—b] [—D*string*] *file1 file2*
diff [—b] [—cefj] [—l] [—L] [—s] *file* [...] *dir2*

DESCRIPTION

If both arguments are directories, *diff* sorts the contents of the directories by name, then runs the regular file *diff* algorithm (described below) on text files which are different. Binary files which differ, common subdirectories, and files which appear in only one directory are listed.

Options for comparing directories:

—**d** Causes *diff* to compare files beginning with '.'. Ordinarily such files are ignored. Even with this option, "." and ".." are ignored.

—**l** Long output format; each text file *diff* is piped through *pr*(1) to paginate it, other differences are remembered and summarized after all text file differences are reported.

—**L** Causes *diff,* in the case of diffing two directories, to unlink dir2////file and link dir1////file to it if the two files are identical.

—**q** Causes *diff* to omit reporting the actual differences between ascii files, and instead merely mention that there is a difference.

—**r** Causes application of *diff* recursively to common subdirectories encountered. Special files are not read with this option; if their major and minor device numbers are the same they are considered to be equivalent.

—**s** Causes *diff* to report files which are the same, which are otherwise not mentioned.

—**S***string*
starts a directory *diff* in the middle beginning with file *name.*

When run with one or more file arguments against a directory, (the "diff file ... dir2" format), *diff* behaves essentially as with the two directory option by constructing a "pseudo directory" from the named arguments. Unfortunately, the named files must be from the same directory. *Dir2* must be a different directory from that which the files are in, of course.

With this option, the -r (recursive) flag is disallowed, the -Sstring option is not supported, and all explicitly listed "." files are compared.

When run on regular files, and when comparing text files which differ during directory comparison, *diff* tells what lines must be changed in the files to bring them into agreement. Except in rare circumstances, *diff* finds a smallest sufficient set of file differences. If neither *file1* nor *file2* is a directory, then either may be given as '—', in which case the standard input is used. If *file1* is a directory, then a file in that directory whose file-name is the same as the file-name of *file2* is used (and vice versa).

There are several options for output format; the default output format contains lines of these forms:

n1 a *n3,n4*
n1,n2 d *n3*
n1,n2 c *n3,n4*

These lines resemble *ed* commands to convert *file1* into *file2*. The numbers after the letters pertain to *file2*. In fact, by exchanging 'a' for 'd' and reading backward one may ascertain equally how to convert *file2* into *file1*. As in *ed*, identical pairs where *n1* = *n2* or *n3* = *n4* are abbreviated as a single number.

Following each of these lines come all the lines that are affected in the first file flagged by '<', then all the lines that are affected in the second file flagged by '>'.

Other options:

—b Causes trailing blanks (spaces and tabs) to be ignored, and other strings of blanks to compare equal.

—t Causes *diff* to print a tab after the '<', '>', and '¦' characters in a difference list, instead of the normal space.

The following options are mutually exclusive:

—c Produces a diff with lines of context. The default is to present 3 lines of context and can be changed, e.g to 10, by **—c10**. With **—c** the output format is modified slightly: the output beginning with identification of the files involved and their creation dates and then each change is separated by a line with a dozen *'s. The lines removed from *file1* are marked with '–'; those added to *file2* are marked '+'. Lines which are changed from one file to the other are marked in both files with '!'.

—D*string*
Causes *diff* to create a merged version of *file1* and *file2* on the standard output, with C preprocessor controls included so that a compilation of the result without defining *string* is equivalent to compiling *file1*, while defining *string* will yield *file2*.

—e Produces a script of *a*, *c* and *d* commands for the editor *ed*, which will recreate *file2* from *file1*. In connection with **—e**, the following shell program may help maintain multiple versions of a file. Only an ancestral file ($1) and a chain of version-to-version *ed* scripts ($2,$3,...) made by *diff* need be on hand. A 'latest version' appears on the standard output.

```
(shift; cat $*; echo '1,$p') | ed — $1
```

Extra commands are added to the output when comparing directories with **—e**, so that the result is a *sh*(1) script for converting text files which are common to the two directories from their state in *dir1* to their state in *dir2*.

—f Produces a script similar to that of **—e**, not useful with *ed*, and in the opposite order.

—h Does a fast, half-hearted job. It works only when changed stretches are short and well separated, but does work on files of unlimited length.

—j Causes a "joint" listing of the input files. Lines common to both files are indicated by an initial '¦' character in the output; '<' and '>' characters begin lines unique to one of the input files, as usual.

FILES

/tmp/d?????
/usr/lib/diffh for **—h**
/bin/pr

SEE ALSO

cc(1), cmp(1), comm(1), diff3(1), ed(1).

DIAGNOSTICS

Exit status is 0 for no differences, 1 for some, 2 for trouble.

BUGS

Editing scripts produced under the —**e** or —**f** option are naive about creating lines consisting of a single '.'.

When comparing directories with the —**b** option specified, *diff* first compares the files ala *cmp,* and then decides to run the *diff* algorithm if they are not equal. This may cause a small amount of spurious output if the files then turn out to be identical because the only differences are insignificant blank string differences.

In the multi-file arguments vs. directory mode ("diff file [...] dir2"):

The algorithm for determining that multi-file arguments all come from the same directory is naive: "./foo" and "foo" are not considered to be equal.

The -Sstring flag is silently ignored.

NAME

tar — tape archiver

SYNOPSIS

tar key [name ...]

DESCRIPTION

Tar maintains an archive of multiple files on a single file (or device). *Tar* was originally designed for archiving on magnetic tape, but can be and often is used on other media, such as floppy disk and networks, as a means of moving groups of files at once. *Tar*'s actions are controlled by the *key* argument. The *key* is a string of characters containing at most one function letter and possibly one or more function modifiers. Other arguments to *tar* are file or directory names to operate on. A directory name argument refers to the directory and all files and (recursively) subdirectories of that directory.

Tar writes all diagnostic and listing information to the standard error output.

A *tar* archive is a stream of 512-byte header structures each of which may be followed by file data filled out with null bytes to the next 512-byte boundary. The end of the archive is signaled by two header structures beginning with null bytes. When *tar* has seen the end of the archive, it will attempt up to two more reads in search of end-of-file so that if it is reading from an archive on magnetic tape, the tape will be positioned after the end-of-file mark, i.e. at the beginning of the next tape file, if any.

The function portion of the key is specified by one of the following letters:

r Write the named files on the end of the archive. The **c** function implies this.

x Extract the named files from the archive. If a named file matches a directory whose contents had been written onto the tape, this directory is (recursively) extracted. The owner, modification time, and mode are restored (if possible). If no file arguments are given, the entire contents of the archive are extracted. Note that if the same file name occurs more than once in the archive, each duplicate one overwrites the previous one.

X *directory*

Like x, plus: takes a *directory* argument as the root of a directory tree for comparison. For each file to be extracted, if it is identical to the file in the corresponding position in the comparison tree, the existing file in the comparison tree is linked to the new file. Otherwise, the new file is extracted as a separate new file as usual.

t List the named files from the archive. The names of the specified files are listed each time they occur on the archive. Archived directory names are listed with a '/' appended to them. If no file arguments are given, all of the names on the archive are listed.

u Add the named files to the archive if they are not already there or have been modified since last put on the archive.

c Create a new archive; writing begins on the beginning of the archive instead of after the last file. This command implies **r**.

C Compare files on the archive against existing files. For each specified file, print a line with a key character followed by the file name.

- L linked to an earlier file on the archive
- S symbolic link
- ? can't read the disk file, so can't compare
-) disk file doesn't exist
- = files compare
- ! files don't compare

The following characters may be used to modify the action of the above function letters.

o Do not write out directory information headers. When writing archives, *tar* normally writes these out, so that owner and modes information about directories is stored in the archive. Former versions of tar don't do this, and when encountering this information on extraction, will give an error message of the form
"⟨name⟩/: cannot create".

p Restore files to their original modes, ignoring the present *umask*(2). Setuid and 'sticky bit' information will also be restored to the super-user.

0, 1, 4, 5, 7, 8
This modifier selects a tape drive device name, "/dev/rmt*x*", where the *x* is replaced by the given number.

v Verbose. Normally *tar* does its work silently. The **v** option makes *tar* type the name of each file it treats preceded by the function letter. With the **t** function, the verbose option gives more information about the archive entries than just their names.

w Print each action to be taken followed by the file name, then wait for user confirmation. If a word beginning with 'y' is given, the action is done. Any other input means don't do it.

f *file* Use the next argument as the name of the archive instead of the default, "/dev/tar". If the name of the file is '—', *tar* writes to standard output or reads from standard input, as appropriate. Thus, *tar* can be used as the head or tail of a filter chain. *Tar* can also be used to move hierarchies with the command

```
cd fromdir; tar cf — . ¦ (cd todir; tar xf —)
```

A similar construction can be used with *rsh*(1) to move hierarchies across a local network.

b *number* Use the next argument as the block size (in units of 512 bytes) for writing to magnetic tape. The default is 20 (the maximum), which gives a block size of 10240. This option should only be used when writing to raw magnetic tape devices. The block size is determined automatically when reading tapes.

l If *tar* can't resolve all of the links to the files written to an archive, it will complain. If the *l* option is specified, it won't.

m Don't restore the modification times when extracting. The modification time will be the time of extraction.

e Press on in the face of garbled header blocks and read errors.

h Follow symbolic links as if they were normal files or directories. Normally, *tar* does not follow symbolic links.

B Forces input and output blocking to 20 blocks per record. (This option was added so that *tar* can work across a communications channel where the blocking may not be maintained.)

R When extracting from an archive, ignore leading slashes on file names, i.e. extract all files relative to the current directory.

U Before extracting each file, unlink any existing file by that name.

If a file name is preceded by —**C**, then *tar* will change directory to that file name. This allows files and directories to be easily assembled onto the archive from various places. For example, to archive files from /usr/include and from /etc, one might use

```
tar c —C /usr include —C / etc
```

Previous restrictions dealing with *tar*'s inability to properly handle blocked archives have been lifted on Berkeley systems thanks to an enhanced magetic tape driver.

FILES

/dev/tar
/dev/rmt?
/tmp/tar*

DIAGNOSTICS

Complaints about bad key characters and read/write errors.
Complains if enough memory is not available to hold the link tables.
Complains and exits if you try to write to an archive specified with the **f** option which is in /dev and doesn't exist or is a regular file.

BUGS

There is no way to ask for the *n*-th occurrence of a file.
The **u** option can be slow.
File name length is limited to 100 characters.
There is no way to selectively follow symbolic links.
The data for a file with multiple links is output to the archive with the first link encountered; so, an attempt to extract a subsequent link by itself will not have the desired result.

Tunis : A Distributed Multiprocessor Operating System

P. Ewens
D.R. Blythe
M. Funkenhauser
R.C. Holt

Computer Systems Research Institute
University of Toronto
Toronto, Ontario, Canada

ABSTRACT

Tunis (Toronto UNIversity System) is a Unix†-compatible operating system. Tunis is a structured, hierarchical system which emphasizes modularity and portability. It is written in a Pascal-like language called Concurrent Euclid. Tunis is object code compatible with Unix, and most programs compiled under Unix may be run without modification under Tunis. The exceptions are programs such as *ps* which make assumptions about the format of the internal data structures of Unix.

Multiprocessor Tunis is a homogeneous tightly-coupled shared memory system. The target architecture consists of several off-the-shelf single board microprocessors connected together via a standard backplane bus. In addition a separate memory board residing on the backplane bus is required to provide global shared memory. Each processor board has its own local memory and access to global shared memory. Shared memory is only used to hold system wide data such as process tables and disk buffers. User programs execute entirely within local memory and may not share writable memory segments. A copy of the operating system also resides in local memory and each processor is capable of concurrently executing it. Multiprocessing is completely transparent to users as the operating system is responsible for processor scheduling. The migration of the operating system and user programs onto local memory sufficiently reduces the interprocessor bus bandwidth requirement to allow processors to execute without on-board hardware caches or specialized busses. This enables the use of economical standard board-level products.

Currently, Tunis runs on several National Semiconductor NS32000 single board computers, coupled via a standard Multibus (IEEE-796) backplane. Each processor features a full NS32000 chip set, two serial ports and up to one megabyte of local memory. Tunis has been tested with up to four processors and provides incremental performance improvements for nominal cost through the addition of single board processors.

† Unix is a trademark of AT&T Bell Laboratories

1. Introduction

In recent years the demand for computational power has outstripped advances in computer technology. A promising alternative is to construct powerful multiprocessors out of a collection of the new generation of microprocessors.

The principal obstacle to the rapid development and commercial use of multiprocessors has been the cost associated with the development and manufacture of specialized hardware to support multiprocessing.

Recently several multiprocessor Unix systems have been developed. Some systems use only two processors [Goble 81] [Bach 84], while others which may use more, generally rely on specialized hardware caches and high speed busses to maintain performance [Finger 85]. An alternative is to make use of local private memories to reduce interprocessor bus bandwidth requirements. If this bandwidth can be sufficiently reduced then standard board-level processors can be used, dramatically cutting the cost of multiprocessor systems

This paper presents a version of the Tunis operating system which is designed to support several processors on a standard speed backplane by utilizing local private memories. Incremental performance improvements can thus be bought for nominal cost (about $3000), by adding additional processors to the system. A basic multiprocessor system with a single processor is no more costly than a normal uniprocessor system.

The Tunis operating system is a structured portable system that is object code compatible with Unix. The goal of the Tunis project has been to adhere to the principles of software engineering such as modularity and information hiding. The principle method of achieving these goals has been to write Tunis in a high level language called Concurrent Euclid. The Concurrent Euclid module construct provides enforced data encapsulation, while the compiler enforces strong type checking. Within Tunis, control is passed strictly hierarchically between modules. Machine dependencies are isolated in separate sub-modules. The resulting structure is both easily understood and highly portable.

2. Concurrent Euclid

Concurrent Euclid (CE) is a high performance Pascal-like systems language [Holt 81]. It is descendent from the Euclid language [Lampson 77] which was designed for developing verifiable systems software. CE omits some of the more complex features of Euclid (notably parameterized types) and adds new features needed for developing concurrent systems software. CE features a small, fast portable compiler, with a replaceable code generator. Experience has shown that a high quality production code generator can be developed in 2 to 4 man-months. Currently code generators exist for the PDP-11, VAX, MC68000, M6809, NS32016, IBM System/360 and Intel 8086. The portability of Tunis relies largely on the portability of the CE compiler.

Concurrent Euclid's concurrency features are based on Hoare style monitors [Hoare 74]. Mutual exclusion is enforced by the monitor construct, and process synchronization is provided by the *signal* and *wait* language primitives. CE provides true machine independent logical processes, in contrast with the pseudo-processes used in most Unix implementations.

CE's concurrency primitives are supported by a small, fast assembly language *Kernel*. Calls to this Kernel are emitted in line by the CE compiler. Porting Tunis requires that this assembly language Kernel be re-written. Since CE provides true logical processes which do not rely on setting interrupt priority levels, the CE support Kernel can be replaced by a Kernel that supports multiple processors without affecting the correctness of the system.

A multiprocessor Kernel must support language primitives across processors. For example, implementation of the *signal* primitive may require that a process wake up another process executing on a different processor. In order to support these language primitives across processors, each processor must be capable of interrupting any other processor.

The entire Tunis operating system is re-entrant except for the machine language Kernel. One of the functions of the Kernel is to support mutual exclusion in CE, yet the Kernel itself must be protected from concurrent execution by more than one process, as its data (i.e queues of runnable and blocked processes) must always be left in a consistent state. In a single processor environment mutual exclusion in the Kernel can be obtained by disabling interrupts when a process enters the Kernel. In the multiprocessor case, the situation is more complicated.

In order to execute the multiprocessor Kernel a processor must first obtain possession of the Kernel. Kernel possession is gained by having processors execute an atomic *test-and-set* operation on an agreed upon flag. Processors finding the Kernel occupied periodically retest the flag until it is reset by a processor relinquishing control of the Kernel.

Processors execute in the Kernel for only a few machine instructions at a time. Thus a processor which finds the Kernel occupied, can expect to gain possession of the Kernel in a very short time.

3. Internal Structure of Tunis

The internal structure of Tunis, is entirely different from that of Unix. Tunis consists of five main layers, each of which is implemented by a CE module [Blythe 84]. The assembly language support Kernel is linked with the CE modules to produce an executable image. The five main modules are : the *User manager*, the *File manager*, the *Memory manager*, the *Device manager*, and the *Utility layer*. Within Tunis, control is passed between modules on a strictly hierarchical basis.

The top-most layer is the User manager. The User manager implements the interface to user programs. This module contains a collection of system processes called *envelopes*. Each envelope is responsible for interfacing to a single user program and carrying out its requests for system services. To do this, each envelope runs the user program by calling a special routine in the Kernel. When a user program performs a system call or its time slice expires control returns to the envelope which then carries out the user program's requests for services by calling the lower levels of Tunis. The User manager also contains a sub-module called the *Family manager* which controls the relationships between user processes and the sending and receiving of Unix signals.

The next major layer is the File manager. It is composed of four internal modules: the Tree File manager, the Flat File manager, the Inode manager and the Cache manager. The File manager is completely machine and device independent. Its function is to implement the Unix file system including pipes, regular files, and the mounting of disk volumes. The Tree File manager implements the hierarchical file system. It is responsible for path name translation. The Flat File manager is responsible for the maintenance of the open file table and for the implementation of pipes.

The Inode manager implements the Unix flat file system. It is responsible for the creation, deletion, expansion, and physical layout of disk files. The Inode manager passes all requests for physical I/O to the Cache manager. The Cache manager implements buffered disks by maintaining an in core pool of recently used disk blocks.

The Memory manager implements the virtual memory abstraction. Layers of Tunis above the Memory manager refer to virtual addresses, while layers below the Memory manager refer to physical addresses. The Memory manager is responsible for the management of physical memory and for the allocation of user address spaces. All transfers to and from user program address spaces are performed by the Memory manager as it has sole knowledge about the physical layout of user programs. There currently exist two versions of the Memory manager; a swapping Memory manager, and a demand paged virtual Memory manager. Outside the Memory manager it is immaterial whether user address spaces are supported by swapping or by paging.

The Device manager is responsible for the management of all physical devices. It contains a sub-module for each major device type, and a router to pass requests to the appropriate

individual device manager. Within each device manager there is a machine independent module which implements the logical device, and a machine dependent driver module which interfaces to the physical device. Device driver processes are synchronous with their devices.

The Utility layer provides ancillary services such as clock management for the rest of Tunis. The assembly language Kernel implements the CE language primitives and provides routines to interface to hardware interrupts. The Kernel is also responsible for low-level process scheduling. In Tunis, all hardware interrupts are hidden in the Kernel. Interfacing to interrupts is done via Kernel primitives which allow a device driver process to wait for a specific interrupt.

4. Target Architecture

4.1 Hardware Configuration

Tunis is designed to support several processors using economical off-the-shelf products. Tunis is targeted for high performance single board microprocessors plugged into a standard backplane bus. The current implementation uses a number of locally designed processor boards [TIL 84] connected via a Multibus (IEEE-796) backplane. Each processor board contains a National Semiconductor NS32000 chip set consisting of a 32016 processor, 32082 memory management unit, 32018 floating point processor, 16201 interrupt control unit and 16202 timing control unit [National 83]. Each processor contains up to 1M byte of local RAM, and two RS232 serial ports. In addition, the prototype system has 1M byte of global (Multibus) memory, a 120M winchester disk drive, and a floppy disk drive.

The NS32016 processor provides a 16M address space. Each processor board has between 256K and 1M of local memory, which the processor can address directly starting at physical address zero. The remainder of the physical address space is mapped onto on board I/O devices and the Multibus. The local memory of each processor also appears in a unique 1M address space slot of the Multibus. The Multibus address of each processor's local memory is selectable by programmable array logic (PAL) used for off-board address decoding.

A simple special purpose board containing about one dozen TTL IC's allows each processor to generate any one of eight system-wide Multibus interrupts by reading specific addresses.

4.2 Software Configuration

In Concurrent Euclid there are two storage classes of data. The first class is module or static data which is allocated at compile time. Module data is accessible to any process executing within the scope of its declaration. The second class of data is stack or temporary data which is allocated at run time and is accessible by only one process.

System calls from user programs are serviced by the envelope processes calling down to the lower layers of Tunis. Since each envelope supervises one user process the majority of per process system information is allocated on the envelope's run time stack. Thus a large portion of data references in the code of the Tunis operating system are to stack variables.

The Tunis operating system contains a fixed number of system (CE) processes. Memory for the stack segments of these processes is allocated by the Kernel at boot time. By default CE processes are allocated to processors on a round robin basis at initialization time. Device driver processes may ask to be located on a specific processor in order to be able to field a local interrupt. Once allocated, a system process must always run on the same processor, since it will accumulate stack temporaries in local memory and thus may not be moved to another processor.

In addition to memory for stack segments, each processor's local memory contains a complete copy of the text segment of the operating system (about 90K bytes). System-wide module (static) data resides in global memory. Thus, with both the stack and text segments of the operating system in local memory, the required interprocessor bus bandwidth is much smaller than each processor's memory bandwidth.

5. Processor Management

In Tunis, user programs interface to the system through the envelope processes located in the User manager. The fixed allocation of CE processes to processors, coupled with the desire to have user programs execute out of local memory, implies that once created, a user program will always execute on the same processor.

The User manager's envelopes queue for new user programs in the Family manager submodule. An envelope can receive a new user process either as the result of a *fork* or *exec* system call.

When a new user process is created as the result of a fork system call, system information to be inherited by the child process is copied into a record in the Family manager, and an envelope running on the same processor is then signaled to proceed. The signaled envelope then copies the inherited information into a local record called the *envelope state*, which stores the user process's state information. The envelope then proceeds to run the user program and service its system calls. Child processes are always forked onto the same processor on which the parent process resides to avoid transferring the image of the program across the interprocessor bus. Low overhead fork schemes such as Berkeley's *vfork* and System V's copy-on-write are easier and more efficient to implement if the child process is forked onto the same processor as the parent process.

When the child process performs a successful exec system call, the system determines which processor the child should run on. All inherited system information is copied into the Family manager, arguments and environment strings are copied into system buffers and the old program's memory partition is destroyed. An envelope which resides on the processor on which the child is to execute is then signaled. This envelope copies the state information back into the envelope state, creates a new memory partition in local memory, reads the executable program off the file system into local memory, copies arguments and environment strings into the new program's memory partition, and proceeds to execute the user program.

Tunis determines which processor a new user program will run on by monitoring the average number of runnable user processes on each processor. If one processor's load is significantly below the average load on all processors then it will receive the next user program that is exec'ed. Otherwise assignment of user processes to processors proceeds on a round-robin basis.

Tunis's scheduling algorithm can be overridden by using a variant of the exec system call which allows the user to specify on which processor a program is to run.

A typical compilation will involve between three and seven scheduling decisions, as a compile is usually broken into several separate programs. The coarse level of processor scheduling in Tunis will result in suboptimal processor utilization when processor loads become unbalanced. For example if there are two compute bound jobs running on each of two processors, both jobs on one processor may finish while the others are still running, leaving one processor idle. However, under typical workloads there are always new programs being exec'ed and old ones exiting, allowing the scheduling algorithm to eventually balance the processor loads.

The exact form of the scheduling criteria is still the subject of investigation, however, preliminary results indicate that the simplicity of the present algorithm and its ease of implementation justifies this coarseness of processor scheduling in Tunis.

6. Memory Management

Physical memory in Tunis is organized as several contiguous regions of local memory. Each processor has its own local memory which has a different base address when viewed from the interprocessor bus. Furthermore each local memory contains an independent set of user programs. The logical choice for memory management is to have an independent demand pager for each local memory.

In order to do this, the module data for the Tunis memory manager is mapped from global memory into a region of local memory at boot time via the system's page tables. Thus the pager's data structures, such as the array of memory frame descriptors, become local private data structures. There is one paging daemon per processor.

The use of local memory results in memory fragmentation. Each pager is given about 870Kb of memory when each processor is configured with 1Mb of on-board RAM. The performance of very large programs would be degraded by trying to run them in a limited amount of memory. In order to try and mitigate the effect of memory fragmentation a significant region of the global memory is devoted to a paging cache. All requests for paging I/O are routed through this pseudo-device. The paging cache is designed to improve the performance of large programs by maintaining a significant number of disk pages in RAM.

7. Device Management

In Tunis, any processor can execute the device managers. If a device manager contains a separate driver process, the processor on which the driver process executes will field the device's interrupt.

Processes executing on any processor can access synchronous (e.g. disk) devices. For example, a system process carrying out a disk request first enters the *disk monitor* which ensures that the process has exclusive access to the controller. At this time disk interrupts are enabled on that processor and the disk request is issued. All other processors have their disk interrupts disabled. When the controller raises the interrupt, it will be fielded only by the processor that contains the process that issued the disk request. Once fielded, disk interrupts are again disabled.

Such device handling methods assure that only the correct processor is interrupted by each device, and allow drivers to be distributed among the processors to ease the interrupt load on a single processor.

8. Performance

The principle factor in limiting the throughput of multiprocessor Tunis is contention for the interprocessor bus. Since this bus is used only to gain access to system data, programs which spend all of their time in user mode will not place any load on the bus. Thus if all programs are executing entirely in user mode, an N processor Tunis will achieve N times the throughput of a single processor Tunis.

As the proportion of time spent executing in supervisor mode increases, the resulting load on the bus increases. It is estimated that, 15-20% of the memory cycles generated while executing in supervisor mode require use of the interprocessor bus. Due principally to the time required for hardware arbitration, the interprocessor bus is up to 50% slower than the on-board bus. Thus, each processor will use about 30% of the interprocessor bus bandwidth while executing in supervisor mode, depending on the type of system work done.

For typical multiuser Unix workloads, up to 40% of the time is spent in supervisor mode [Bach 84]. This implies that each processor uses up to 15% of the interprocessor bus bandwidth averaged over time spent both in user and supervisor modes. There are, however, a large number of programs which spend a fairly small fraction of their time in supervisor mode. Examples include compilers, text processing and simulation programs. An N processor distributed Tunis running such a job mix can be expected to deliver nearly N times the throughput of a single processor.

Measurements on a dual processor system, indicate that with balanced processor loads the system achieves throughput of about 1.9 times that of a uniprocessor version when 20% of the time is spent in supervisor mode, and a throughput of about 1.7 when 40% of the time is spent in supervisor mode.

Contention for execution permission for the machine language Kernel is not expected to be a limiting factor to the throughput of Tunis. Measurements on a loaded system indicate that the Kernel is occupied about 15% of the time the processor is executing in supervisor mode. The current compiler generates code to issue calls to the Kernel to implement concurrency primitives. An enhanced compiler would generate code to perform certain concurrency primitives in-line thus reducing Kernel occupancy to about 8%. Since each processor typically spends considerably less than half its time executing in supervisor mode, the Kernel is expected to be occupied about 3N% of the time, where N is the number of processors in the system.

9. Conclusions

Although still under development, the distributed version of the Tunis operating system promises to provide an effective method for structuring tightly-coupled multiprocessor systems.

References

[Bach 84]
Bach M.J., Buroff S.J. "A Multiprocessor Unix System". *USENIX Conference Proceedings*, August 1984, pp. 174-177.

[Blythe 84]
Blythe, D.R., Ewens P., Funkenhauser M., Hume M., Holt R.C. *The Structure of the Tunis Operating System*. CSRI technical note, Computer Systems Research Institute, 1984.

[Finger 85]
Finger E.J., Krueger M.M., Nugent A.F. "A Multi-CPU Version of the Unix Kernel - Technical Aspects and Market Need". *USENIX Conference Proceedings*, January 1985, pp. 11-22.

[Goble 81]
Goble, G.H. "A Dual Processor Vax 11/780". *USENIX Conference Proceedings*, September 1981, pp. 114-138.

[Hoare 74]
Hoare, C.A.R. "Monitors: An operating system structuring concept". *Communications of the ACM*, 17(10), October 1974, pp. 549-557.

[Holt 81]
Holt, R.C., Cordy, J.R. *Specification of Concurrent Euclid*. Report CSRG-133, Computer System Research Group, 1981.

[Holt 83]
Holt, R.C. *Concurrent Euclid, Unix and Tunis*. Addison-Wesley, Mass., 1983.

[Lampson 77]
Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G., Popek, G.J. "Report of the Programming Language Euclid". *ACM SIGPLAN Notices*, 12(2), February 1977.

[National 83]
National Semiconductor. NS16000 DataBook, 1983.

[Popek 84]
Popek, G. et. al. "The LOCUS Distributed Operating System". *USENIX Summer Conference Proceedings*, August 1984, pp. 49-70.

[Ritchie 74]
Ritchie, D.M., Thompson K. "The Unix time sharing system". *Communications of the ACM*, 17(7), July 1974, pp. 365-375.

[TIL 84]
TIL Systems Ltd. ZP1632 Multibus Processor Board Reference Manual, July 1984.

VLSI Assist in Building a Multiprocessor UNIX System

Bob Beck
Bob Kasten

Sequent Computer Systems
14360 NW Science Park Drive
Portland, Oregon 97229

ABSTRACT

Multiprocessors have been of interest to computer scientists and designers since the first computers. Three factors have limited the commercial success of multiprocessor systems; entry cost, range of performance, and ease of application. Recent advances have removed these limitations, making possible a new class of multiprocessor systems based on VLSI components.

A set of requirements for constructing an efficient multiprocessor system are detailed, including: low-level mutual exclusion, interrupt distribution, interprocessor signaling, process dispatching, caching, and system configuration. A system that meets these requirements is described and evaluated.

1. Introduction

From the point of view of an application program, a computer system provides one or more processing **engines** to execute application code. A computer that provides exactly one such engine is a **monoprocessor.** A computer providing two or more such engines is a **multiprocessor.** Note that this application program viewpoint makes the classification independent of other, special purpose processors which do not directly execute application programs (e.g., intelligent slave I/O devices, smart disk controllers, buffered terminal multiplexers, etc).

Multiprocessor systems typically come in two basic forms, depending on how system memory is distributed:

- A **multi-computer** system provides multiple, possibly different, processing engines. Typically, application programs are statically placed on these engines and cannot migrate between them. Multi-computer systems typically provide a per-processor memory, and often some (smaller) amount of memory addressable by all processors for operating system functions.
- A **shared-memory multiprocessor** system provides multiple processing engines, typically identical (or nearly so). Application programs run in a single memory shared among all processors, and so have the potential to execute on any processor at any time.

Shared-memory multiprocessor systems can be further classified according to how the operating system is distributed among the processors:

- In a **master-slave** (or **asymmetric**) multiprocessor, the operating system runs on a special, preselected processor. All operating system code (system services, interrupt handling, trap handling, etc.) runs on this processor. If necessary, a process is context-switched to the **master** processor when making use of the operating system. Such systems have the

advantage of providing multiple processing engines for application programs and being easier to construct from a software point of view (since there is minimal change from a monoprocessor operating system). However, master-slave systems only provide performance benefit if the workload is user-mode intensive; they can easily bottleneck when operating system use increases. As a result, such systems are typically limited in the number of application processors they support [Gobel 1981]. In addition, the entire system fails if the master processor fails.

- A **symmetric** multiprocessor makes no distinction among its processors. Processors are viewed as another resource that the operating system manages. All processors can run all application and operating system code. Processes may be dispatched on any processor in response to process or system needs; in fact, a process often executes on several processors during its lifetime, without being aware of the transition. These systems share the advantages of master-slave systems, but avoid the problems: since all processors can execute system services, no context switch is necessary for system calls or traps; interrupt handling may be distributed among the processors to avoid overloading any single processor. Often these systems can support more (perhaps many more) processors. Also, since there is no master processor, the system can run with some loss in performance if at least one of its processors is functional.

Note that there are many variations on the above classifications. However, they serve to define a working terminology. We use the term **multiprocessor** in the remainder of this paper to refer to a shared-memory symmetric multiprocessor. The Sequent Balance 8000 is an implementation of this kind of multiprocessor system. Further expansion of multiprocessor classifications and a more detailed description of the system are available elsewhere [Fielland 1984].

Building a multiprocessor computer system requires solution to a host of problems not present in the construction of a monoprocessor system. We discuss some of the more interesting of these problems and present our solution.

2. Traditional Multiprocessor Problem Areas

Multiprocessor computer systems introduce a number of problems not present in monoprocessor systems including low-level mutual exclusion, hardware interrupt distribution, inter-processor signaling, additional system configuration issues, process dispatching among processors, cache policies, and pragmatic concerns such as ability to integrate device drivers into the multiprocessor environment. This section discusses these problems in more detail.

2.1 Low-Level Mutual Exclusion

Monoprocessor systems typically handle this problem by observing that there is no actual concurrency in the operating system and the only reentrancy in the operating system is the result of servicing hardware interrupts. Disabling some or all interrupts at appropriate times is sufficient to avoid *concurrent* (actually, reentrant) access to system data structures. This simplification avoids much of the concurrency issue within the operating system. UNIX* on a monoprocessor uses this technique.

A multiprocessor system differs in that it allows more than one process to be executing operating system code at literally the same time; multiple processes may be attempting to access and modify system data structures concurrently. Since there are potentially multiple concurrent threads of execution in the operating system, interrupt-masking techniques fail to fully solve the problem. Explicit control over concurrent accesses to system data structures is required. This control takes the form of **test-and-set** variables and higher-level synchronization primitives built from these.

UNIX is a trademark of AT&T Bell Laboratories.

A test-and-set variable is typically implemented with a hardware mechanism that insures a read/modify/write of the variable is atomic; regardless of the number of concurrent accesses attempted, exactly one at a time will proceed. The variable can be set or cleared; a request to set the variable returns the previous state (set or cleared), and sets it. Using a test-and-set variable to control access to a data structure results in **busy-waiting**; i.e., a process spins in a program loop constantly re-testing the variable for *clear* state. If processes obey an appropriate locking protocol, at most one process can access a given data structure at a time.

Since busy-waiting is wasteful of processor cycles, it is desirable to avoid it whenever possible. Processes should hold test-and-set variables for as short a time as possible. Longer term resource allocation and waiting for events require another mechanism, which allows processes to block (i.e., sleep) until access is possible. Counting semaphores can be built from the test-and-set primitive, and provide the necessary semantics.

It is beyond the scope of this paper to discuss the detailed semantics and higher-level uses of these primitives.

2.2 Interrupt Distribution

Monoprocessor systems deliver interrupts to the single processor in the system. When multiple processors are available, it is desirable to distribute the interrupt load among the processors. This distribution can be static or dynamic; static distribution assigns interrupts to fixed processors at hardware and/or software configuration time. Dynamic interrupt distribution allows any processor in the system to handle any interrupt depending on dynamically changing system load characteristics.

Dynamic distribution of interrupts has several advantages over static distribution:

- There is no complex assignment issue (which interrupts to which processor); the interrupt load is automatically distributed.
- Hardware components need not be tailored to a specific environment (e.g., using interrupt-level jumpers).
- No single processor is overloaded with interrupts due to a poor static assignment.
- It is possible to arrange for idle processors (if any) to accept interrupts, thus automatically off-loading interrupt processing from processors doing useful work.
- Interrupts may be fielded by processors executing lower-priority processes.
- Processors may be removed from service without removing the server for any hardware interrupt source.

2.3 Inter-Processor Signaling

It is sometimes necessary for one or more processors in a multiple processor environment to signal another. Examples include:

- Telling a processor to re-dispatch itself, if a higher priority process has become ready to run.
- Delivering a software signal. Delivering the signal may necessitate forcing the signaled process to enter the operating system so it can accept the signal.
- Telling a processor to remove itself from service. The processor must cleanly stop executing a process (if necessary) and turn itself off.
- Initiating low-priority interrupt services such as network protocol routines, lower-priority time-of-day clock services, time-slicing, etc.

Another, less likely, occurrence is a system panic, induced by a hardware- or software-detected inconsistency in the system's behavior or state information. To cleanly shut down the system, the processor that sensed the condition must force other processors to stop executing.

2.4 Process Dispatching

Ideally, in an N processor system the N highest priority processes are executing at any point in time. If fewer than N processes are executable, the remaining processors are idle and immediately available to run processes.

When a process context-switches, it may next execute on a different processor than before the context-switch. This results in the cache of the new processor having little or no useful context for the process. Since filling a processor cache places a burden on the system bus it is important to avoid unnecessary context-switches, as these impact overall system performance. Of course, an unnecessary context-switch also wastes processor cycles.

These goals are difficult to achieve in practice, since dispatching decisions are made using a snapshot of system state information and this state may change before the results of the decisions are realized. A heuristic algorithm is used to keep the right set of processes running, while avoiding unnecessary context-switches whenever possible.

Since the number of processors can be less than the number of processes that want to run, some form of time-slicing may be used to multiplex processors as in a time-shared monoprocessor system. The time-slice algorithm can distribute the time-slice load among processors running the lowest priority processes. This also helps reduce the number of context-switches.

2.5 System Configuration

To help reduce complexity and cost of running a multiprocessor system, it is desirable that a single configuration of an operating system be capable of running on as many hardware configurations as possible. This allows a single set of software to run on a variety of machines, greatly simplifying the maintenance problem. Although this problem is not new, the presence of multiple processors in the system aggravates it. Additionally, to avoid the requirement of source code for configuration (as required in 4.1bsd), the system must be configurable from a binary-only copy.

A mechanism is needed to recognize the various major hardware components in the system (i.e., processors, memory, I/O controllers). Ideally, this mechanism is independent of the particular type of hardware, as this simplifies the system initialization code. Whenever possible, the operating system should allocate data structures to represent these hardware components during system initialization rather than compiling in hard-coded constants. For example, a single operating system binary should boot and run on a system with any number of processors. This simplifies binary configuration and makes the software more independent of particular hardware configurations.

An additional issue is determining which hardware components are statically configured (i.e., at boot-time) and which are dynamic (i.e., at run-time). To assist in debugging and system tuning, processors should be dynamically configurable (i.e., brought in and out of service) during normal system operation. For example, it is often easier to debug a device driver in a single-processor environment and then test the multiprocessor aspects of the driver after it is known that the driver functions properly with the hardware. Dynamically altering the amount of system memory and I/O controllers are less well motivated.

2.6 Main Memory Cache

Even in monoprocessor systems the system bus typically imposes an access protocol that increases the processor's latency to main memory. The main memory may not be fast enough to meet the processors demands even if the bus had no overhead, since memory components fast enough for

the processor are too expensive to construct the large system memory. As a result, monoprocessor systems often employ a main-memory cache to decrease the processor's latency to its more frequently accessed instructions and data. The cache provides a small amount of very fast memory, between the processor and the system bus, that can keep up with the processor's demands.

A multiprocessor system increases the need for a cache since there are multiple processors concurrently executing. Often the bus is designed with bandwidth to exceed the needs of an individual processor, but cannot meet the demand that multiple processors create. Additionally, multiprocessor systems frequently employ spin-loops for low-level mutual exclusion primitives and idle-loops; if these loops execute entirely out of processor-local cache, the system bus bandwidth is not degraded.

A number of cache policies are documented in the literature [Goodman 1983]. These techniques include write-back, write-through, write-once, and hybrids. A related parameter is whether or not the cache *watches* the system-bus to sense changes in local copies of data. A bus-watching cache simplifies the software and allows low-level spin-loops in the operating system to execute entirely out of cache, relying on the cache to notice the change in value of relevant state variables.

2.7 Monoprocessor Device Drivers

Throughout most if its history, UNIX has executed on monoprocessor systems. In the UNIX kernel, many algorithms take advantage of the assumption that there is only one processor. When moving to a multiprocessor environment, many of these algorithms must be re-evaluated, and modified or rewritten to use more formal techniques (including test-and-set variables and semaphores). All code that is part of the multiprocessor operating system must be evaluated in this manner. Unfortunately, this includes device drivers, which are often written or otherwise supplied by customers of the system as opposed to the developers.

Systems programmers are accustomed to accommodating the hardware dependencies present in a new environment. For example, a driver that ran in a Unibus* environment must be modified to run in a VAX since the Unibus is accessed differently. The necessary changes for a multiprocessor environment are not as familiar to the UNIX community, however. Thus it is desirable to ease this porting task as much as possible.

One possibility is to insist the system run only a single processor until the driver is fully debugged and tested in the new multiprocessor environment. This is non-optimal in that other processors in the system are not usable during this time, and there may be drivers that don't warrant the investment to bring them to full multiprocessor capability (infrequently used, low throughput, etc.).

A better approach is to accommodate monoprocessor versions of drivers in a multiprocessor environment. This allows the driver to be adapted to the new hardware environment without adding support for multiprocessing, while allowing use of all processors in the system. Ideally, there is no modification to the portions of the operating system that call drivers, and no modification to the driver beyond hardware dependencies. The operating system must arrange that the driver code run on a particular processor, including the driver's interrupt handlers (thus picking up some aspects of asymmetry). If possible, this processor should be chosen at system initialization time to avoid the necessity of any particular processor being present in the system.

Unibus and VAX are trademarks of Digital Equipment Corporation.

3. Cache in a Multiprocessor

Cache memories have long been used to improve performance in traditional monoprocessor architectures. In monoprocessors, this local memory between the processor and main memory is used to reduce bottlenecks in systems with fast processors and relatively slow memory and/or system buses. In a symmetric shared-memory multiprocessor architecture, multiple processors and multiple I/O channels tend to place considerably higher demand on the system bus bandwidth and memory subsystem over that of a monoprocessor architecture. Therefore, reduction of the potential bottlenecks of system bus contention and main memory latency is critical to multiprocessor performance. Avoiding these bottlenecks, in addition to the traditional performance gains, motivate the need for a cache memory to be placed between the system bus and each processor in a multiprocessor.

Per-processor caches minimize the effects of bus contention and memory latency by keeping frequently used instructions and data in the cache. This greatly reduces the system bus traffic. When data is needed from main memory it can be transferred to the cache in larger chunks, making more efficient use of available bus bandwidth. However, multiple caches in a multiprocessor create additional problems. First, multiple caches impose a size and cost constraint. This is addressed through the use of VLSI components. Secondly, there is a potential problem in insuring data consistency, with multiple copies of shared data existing in different caches. With proper cache management policies this problem is averted.

3.1 Cache Effectiveness

As in a monoprocessor the cache effectiveness is measured via its cache hit ratio. A **cache hit** occurs when the processor finds the requested data in its cache. A **cache miss** occurs when the request cannot be satisfied by the cache and the processor must fetch the data from main memory. The **hit ratio** is the number of cache hits divided by the number of memory requests. The cache hit ratio is affected by the following design factors:

- Cache size The hit ratio is greatly enhanced if the cache is large enough to hold looping constructs.
- Cache line size This is the number of bytes fetched from main memory after a cache miss.
- Cache block size This is the number of bytes in a replaceable unit in the cache. Often cache line size equals cache block size.
- Cache management policy There must be efficient mapping of main memory's large physical address space to the small cache space.

The remainder of this section will focus on the cache management policies relevant to the multiprocessor architecture. A more detailed discussion of the cache design tradeoffs is found in [Efland 1984].

3.2 Cache Management Policy

Multiple caches in a system with shared memory creates a potential data consistency problem. This is not a problem on memory reads, but if one processor writes the data, that write must be reflected in all other processors where the data is cached, but now invalid. To prevent processors from using any stale data, each cache controller must recognize when another processor's write operation invalidates any of its cached data. Thus, the data consistency problem must be resolved by the cache controller logic in conjunction with its write policy.

In order to maintain data consistency, all cache controllers continuously monitor the system bus watching for write operations. When a write from another processor occurs to an address that is in its cache, the local cache controller invalidates its copy. If the local processor again references the data, a cache miss will occur and the data will be reread from main memory. This technique is called **bus watching.** For data consistency to be maintained the cache must also be a physical

cache. That is, the value used to address the data in the cache is based on the physical address of the data rather than its virtual address in the process. A virtual address is insufficient since a chunk of shared physical memory may be mapped into different virtual addresses by different processes.

Several write policies have been devised to ensure data consistency efficiently (see figure 1). These include, write-back, write-once, write-through, and various hybrids. The **write-back** technique relies on software to distinguish global data from local data (e.g., the stack segment). Only the global data is immediately updated into main memory and hence made visible to other processors. The local data is updated when the software dictates, most commonly during a context switch. The update can be done by special hardware that makes efficient use of the bus bandwidth. Although write-back reduces bus contention and maintains the consistency of shared data (with software assistance), it is more complex and thus has a higher implementation cost.

Policy	Effect on Bus Traffic	Effect on Overall Performance	Effect on Cost
Write-back	Produces minimum bus traffic since no local data is written to main memory until a block is replaced	Best for bus traffic reduction, but needs software assist	High development cost
Write-once	Less traffic than write through, slightly more than write back	Best performance improvement	High parts cost and complexity
Write-through	Greatest bus traffic	Performance penality largely overcome by write buffer	Medium hardware cost, lower software complexity

Figure 1: Cache Write Policy Design Tradeoffs

Write-once is performed completely in hardware. The first time a processor writes a variable, the cache controller captures the variable and also writes it to main memory. This updates the system global memory and makes the write event visible on the system bus, so that any other caches with a copy of the data can mark it invalid. Subsequent writes to the variable are written only locally to the cache. The cache controller bus-watches as in other techniques, but will also respond to a memory read request *instead* of the memory controller(s) when a locally dirty variable is read by another processor. This form of active bus-watching provides high-performance, software-transparent cache management, but at the cost of high complexity in the cache controller, system bus, and system memory controller.

The technique employed in the Sequent Balance 8000 is called **write-through, with a write buffer.** The write-through policy relies on only the bus watching mechanism to guarantee data consistency. All write operations are made to main memory, ensuring that main memory always contains valid data. Since most memory accesses are instruction fetches, a typical work load performs many more read requests than writes. Since writes are a relatively small percentage of total memory accesses, the performance advantage of the other techniques does not warrant the complexity and implementation cost of those techniques. The impact of writes on processor performance can be reduced by buffering writes, allowing the processor to continue execution asynchronously with the actual write to system memory.

4. System Link and Interrupt Controller

The System Link and Interrupt Controller (SLIC) chip was developed to address a number of the multiprocessor problems. One SLIC chip is coupled with each major component in the system (processor, memory controller, I/O controller), providing support for interrupt distribution, low-level mutual-exclusion, and configuration and error control. A major goal of the SLIC design was to remove many of these concerns from the software, or to provide sufficient hooks to simplify the problems wherever possible. It was also desired to simplify the system bus, to lower the cost while maintaining high performance and reliability. Figure 2 gives a block view of the SLIC. This section describes the functions of the SLIC. The next section describes how the SLIC is used to address several of the multiprocessor problems described previously.

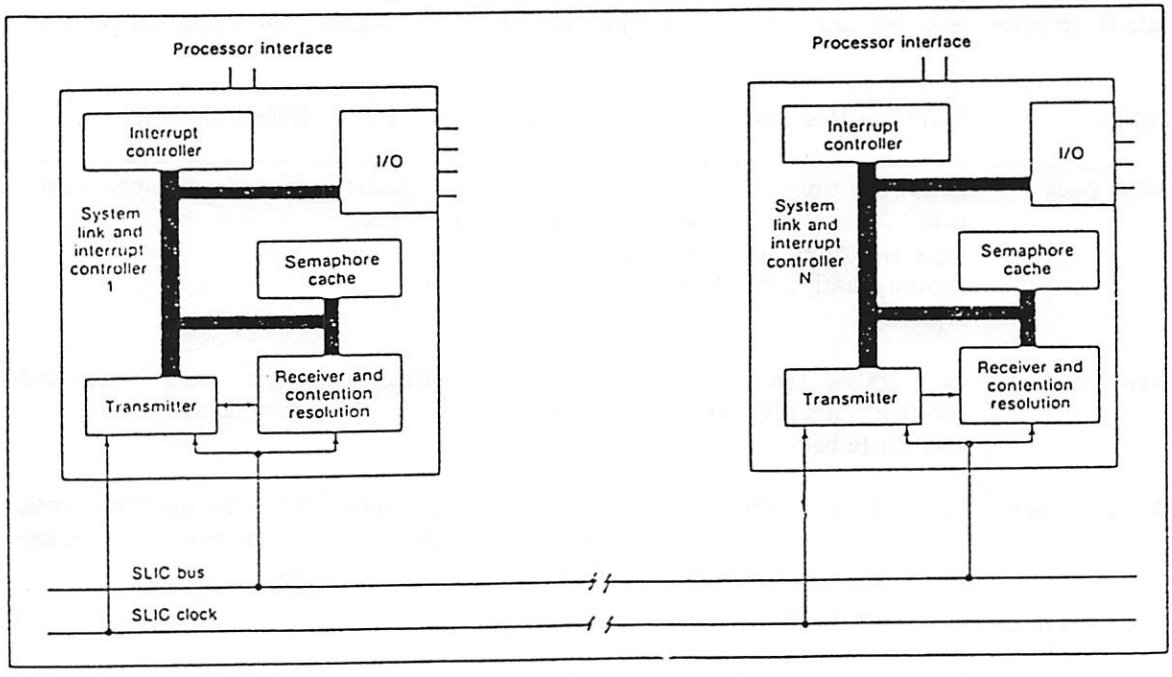
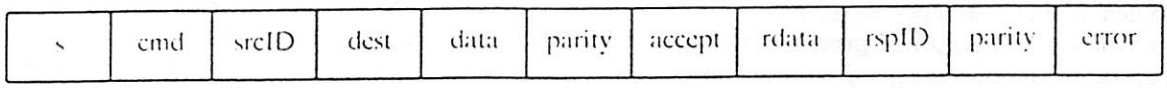


Figure 2: SLIC Block Diagram

4.1 SLIC Messages

SLIC chips in the system communicate with a simple message-passing protocol, using two lines from the system bus for a bit serial, wired-OR medium (the SLIC *bus).* The SLIC bus derives its clock from the system bus but otherwise operates asynchronously; thus, SLIC message traffic is transparent to the system bus, not impacting bus bandwidth. SLIC bus access is arbitrated using wired-OR resolution; in case of contention, the higher priority message automatically wins the arbitration (priority is part of the message encoding). Each SLIC has a system-unique *number,* derived from its host board's position in the system backplane; this number is used to break a tie during arbitration. A SLIC message consists of two parts: request and response, as illustrated in figure 3.

A SLIC request message contains an 8-bit command, source address, 8 bits of message data, and a destination address. The response contains an 8-bit data field and status information. The interpretation of the message and response data is a function of the command; several of these are discussed below. The command values are chosen to reflect their relative priority, allowing



Request Part		Response Part	
s	start bit	accept	interrupt accepted status
cmd	encoded command	rdata	response data
srcID	sender SLIC address	rspID	responder's SLIC address
dest	encoded destination	parity	parity protection
data	8-bit message data	error	SLIC error
parity	parity protection		

Figure 3: SLIC Message Format

the wired-OR arbitration to automatically pick the highest priority message if there is contention. The source SLIC address is appended to the command to resolve arbitration ties when two or more SLICs are trying to send the same command message. The SLIC will automatically retry the message if an attempted send looses arbitration on the SLIC bus. Each SLIC sets a status bit to indicate success or failure in sending the message.

A message addresses one or more SLICs (encoded in the destination field). Each of these SLICs will try to respond to the message if appropriate, by trying to send the response part of the message. It must start responding in the bit-time of the request's parity bit. In the event that more than one SLIC attempts to respond, arbitration is guaranteed by including the SLIC address of the responder in the response. It is possible that no SLIC responds to the message (depending on the command); the sending SLIC records this status in addition to the *message-sent* status. After the message is sent and response received, the software (or other entity controlling the SLIC) uses the status bits to determine the appropriate next action. If the message wasn't received (e.g., because the destination SLIC(s) is ignoring interrupts), the message can be retried. Since proper operation of the SLICs is paramount to system integrity, the system currently panics if any SLIC error is detected (this has never happened).

4.2 SLIC Interrupts

One class of SLIC messages encodes interrupts of various kinds: maskable, non-maskable, and software interrupts can be sent directly to other SLICs or broadcast to a group of SLICs. Interrupts are identified by one of eight **bins**, each corresponding to a processor hardware vector. The bin number is encoded as part of the SLIC command field: bin 7 is the highest arbitration priority, bin 0 is the lowest. The SLIC message data in an interrupt message is used to further subdivide each bin into 256 logical interrupt vectors, allowing interrupt sources to be adequately distinguished. It is stored in a readable SLIC register for the convenience of the operating software.

SLIC provides two basic kinds of interrupt, maskable and non-maskable, corresponding to the interrupt request lines present on most microprocessors. The type of interrupt and bin number are encoded in the command field of the SLIC message. Interrupts are further characterized in how they are directed: to a particular SLIC (**directed** interrupt) or one of a group of SLICs (**group** interrupt; the selection of group is an initialization parameter to the SLIC). All SLICs in a group attempt to accept the interrupt, subject to their current state (more on this below); at most one of the SLICs will accept the interrupt, using the responding SLIC's address for final arbitration resolution. To allow some software control over arbitration in accepting group interrupts, the SLIC supports an 8-bit register whose upper 5 bits are fully software controlled (the lower 3 bits are randomly generated by the SLIC). This value is placed in the *rdata* field of the response part of the message, as illustrated above (there is no actual response data returned for

an interrupt message, so this field can be used for interrupt acceptance arbitration).

When a SLIC accepts an interrupt, it copies the message data into an internal register, and activates the appropriate processor interrupt line. The SLIC then refuses to accept another interrupt until the operating software acknowledges it has started its interrupt handler. This also tells the SLIC it has read the message data (the SLIC currently has only one register to hold maskable interrupt message data). Bin 0 differs in that it will always accept an interrupt, logically OR-ing the message-data into the separate *bin0 message data* register. Software can use this bin to post various kinds of events to selected processors. The current SLIC can be viewed as a subset implementation of a more general model where attributes such as interrupt queuing and message OR-ing are selected on a per-bin basis.

As in monoprocessor systems, there is a need to mask interrupts at various points in time. SLIC supports an 8-bit register, controlled by the processor, in which each bit masks or enables the corresponding interrupt bin for that SLIC. A SLIC with a particular bin masked will not attempt to arbitrate to accept an interrupt in that bin that it might normally accept. Thus, software can arrange that particular sets of interrupts be ignored for specified periods of time. As a result, if a processor or I/O controller attempts to send an interrupt (even a group interrupt), it is possible that the interrupt will not be immediately accepted. This is resolved by having the interrupt source resend the interrupt message. These somewhat non-intuitive semantics make sense in a multiprocessor environment: if a masked SLIC queued the interrupt, it might actually take longer to start servicing the interrupt since another processor may unmask its SLIC before the queuing processor starts service. (It also keeps the SLIC implementation simpler.)

Most operating systems accumulate per-process execution statistics, as exemplified by the UNIX notion of *user* and *system* time reported by the *time* command. On a multiprocessor system, a single system clock could only support this by broadcasting a periodic interrupt and insuring that all processors accept it. It is more convenient to have each processor generate its own clock. This also has advantages with respect to cache usage: per-processor clocks can be mutually asynchronous, which avoids forcing all processors to change their cache context at the same time (by dropping into the kernel clock handler), thus spreading the system bus load more evenly. To support this, the SLIC implements a programmable internal timer that can be connected to any interrupt bin.

4.3 SLIC Gates

Each SLIC implements a set of 64 binary semaphores, called **gates**, and supports a set of SLIC commands to atomically test-and-set them. Each SLIC contains a copy of the value of each of the 64 gates at all times. If a processor attempts to *lock* a gate, that processor's SLIC uses its local copy of the gate value to determine if a SLIC-bus message is necessary. If the gate is already locked, the SLIC sets a status bit and sends no message. If the gate is unlocked, the SLIC sends a message to attempt to lock the gate. This message is seen by all SLICs, who set their local copy of the gate to *locked*. The SLIC *lock-gate* message is arbitrated to insure that at most one SLIC sends a lock-gate message at any point in time, thus assuring the atomicity of the operation. Another SLIC message unlocks a particular gate; all SLICs clear their internal copy of the gate. The gate number is encoded in the request data field of the message.

Since gates are used for low-level mutual-exclusion in the operating system, it is important that any *write-behind* data is really in system memory before actually releasing a gate. For example, data modified while the gate was locked might be in the processor write-buffer on its way to system memory but not yet there. To avoid race conditions that could result, SLIC doesn't actually send the *unlock-gate* message until the processor write-buffer empties (the write-buffer exports a status bit for this purpose). Thus, when software releases a gate, it has automatically guaranteed the consistency of system memory.

4.4 SLIC Slave Registers

Each SLIC supports a 256 byte local address space, called **slave registers**. These registers are read or written by the SLIC itself or by any other SLIC in the system using two messages directed to a particular SLIC: one to specify the address, the next to read or write the data. Each hardware board in the system implements a subset of these addresses for its SLIC.

4.5 SLIC Implementation

The SLIC has been implemented in a 6000-gate custom CMOS gate array component, designed at Sequent. The part was extensively simulated and was fully functional in first silicon. More details on the design and implementation are published elsewhere [Lovett 1984].

The SLIC is accessed via 17 byte-wide locations in its processor's address space. Some of these locations are simple read and/or write registers; for example, local SLIC interrupt mask, arbitration priority register, local SLIC number (provides processor identification as well), timer control registers, etc. Sending a message involves loading several registers with the message data, encoded destination address, and any other necessary data, then loading the command register to start the command. The software then spins looking at a status bit to see when the command completes (very few spins are needed). Once the SLIC request is complete, the status register contains bits describing aspects of the result: was the message sent, was it accepted, were there any errors, etc. Software can retry the command, return a success/fail indication, or whatever makes sense. The programming model relies on spin-waiting for results since each SLIC command completes quickly with its status; servicing an interrupt would impose unreasonable overhead.

The code example in figure 4 illustrates the programming model with a simplified version of the code used to send a maskable interrupt. Code to send a non-maskable interrupt, send a software interrupt, read or write a slave register, or request or release a gate is similar.

5. SLIC Solutions to Multiprocessor Problems

The SLIC subsystem provides a low-cost solution to a number of the traditional multiprocessor design problems discussed above. The SLIC subsystem supports system-wide interrupt control, mutual exclusion primitives, and a mechanism for inter-module communication. This functionality is used to resolve problems regarding processor synchronization, dynamic interrupt distribution among processors, monoprocessor driver support, inter-processor communication, dynamic load balancing of processes among processors, and dynamic system configuration.

5.1 Processor Synchronization

A basic necessity in a multiprocessor system is a fast, efficient synchronization primitive. The SLIC supports this function with a set of 64 single-bit **gates**. Gates are logically equivalent to a test-and-set primitive and are spin-oriented. That is, the process loops requesting the gate until it is acquired. In many machines the test-and-set function is implemented via an interlock signal on the system bus and/or in the memory controller itself. This imposes extra complexity in the system bus architecture. The use of SLIC gates as the synchronization primitive has two main advantages. First, gate operations are via the SLIC bus. Since the SLIC bus is separate and asynchronous to the system bus, SLIC gate accesses do not use system bus or memory bandwidth. Second, since each SLIC knows the status of all gates, the spinning is done watching the local SLIC status register and not across the SLIC bus. Thus SLIC bus bandwidth is not adversely affected. This technique relieves one of the performance bottlenecks found in previous multiprocessor systems when under heavy load. Since the mutual exclusion is achieved via the gate mechanism, the systems bus and memory architecture is simpler. This allows the system bus to be built at a lower cost, with performance maximized, and with a higher degree of reliability.

```
sendinterrupt(dest, bin, data)
{
        /*
         * Set up target of interrupt (dest),
         * and "vector" number (data).
         */

        slic->sl__dest = dest;
        slic->sl__data = data;

        /*
         * Set up Command and Bin Number,
         * wait for interrupt to be sent,
         * and loop until interrupt accepted.
         */

        do {
                slic->sl__cmd = MINTR | bin;
                while(slic->sl__stat & BUSY)
                        continue;
        } while((slic->sl__stat & OK) == 0);
}
```

Figure 4: Send Interrupt Code Fragment

The DYNIX* operating system, Sequent's version of UNIX 4.2bsd, is implemented using three types of mutual exclusion primitives built from gates. These are spin locks, counting semaphores and the direct use of the gates themselves.

The gate is the lowest level primitive and is directly implemented in the SLIC chip. Gate acquisition is the fastest among the three types of mutual exclusion primitives. But since there are a finite number of gates, they are used only in the most time-critical regions. For example, a gate is used to synchronize the processors' accesses to the RunQueue. Gates, because of their spin-oriented nature, are held for only a short period of time to minimize contention. However, when there is contention, no system bus or SLIC bus cycles are consumed by a processor waiting for a busy gate.

Locks are defined to multiplex gates. Whereas there are only 64 gates, the number of locks is unlimited. A lock uses a shared memory location to encode the lock state (locked or unlocked) guarded by a gate. The gate is acquired only to manipulate the lock variable (i.e., to set the lock). Therefore, lock operations are also atomic. A single gate may be used to synchronize accesses to many locks. For example, changes to the state and flag variables for a process in the process table must be performed atomically. This is done by first acquiring a per-process lock. If the system is configured for 500 processes, there will be 500 such locks. However, only 10 gates might be assigned to synchronize the manipulation of those locks. Locks, like gates, are spin-oriented and are used to guard short critical regions. Since each processor is equipped with a bus-watching cache, the instructions and data accessed while waiting for a lock are cached. Therefore, no system bus cycles are consumed waiting for a busy lock.

DYNIX is a trademark of Sequent Computer Systems, Inc.

In a monoprocessor, the SPL (set interrupt priority level) mechanism is used to provide the synchronization necessary between interrupt-level and process-level access to shared data structures. Specifically, the process-level code must raise the interrupt priority level of the processor to block out interrupt routines before executing a critical region. In a multiprocessor architecture, a form of interprocessor synchronization (i.e., gates and locks) is necessary in addition to the SPL mechanism. Process-level code must raise the processor interrupt priority level when acquiring a lock to avoid the deadlock that would occur when an interrupt routine on the same processor attempts to acquire the same lock.

The highest-level mutual exclusion primitive used is the counting semaphore. The counting semaphore consists of a counter and a wait queue. A gate is used to guarantee the atomicity of the semaphore manipulations. Semaphores are used to block waiting for an event, or when the critical region guarded by the semaphore is very long. Instead of all waiters being awakened on an event, only one process at a time is awakened and allowed access to the resource.

Semaphores completely replace the conventional UNIX sleep/wakeup mechanism. In the sleep/wakeup model, all waiters are awakened and they all compete for the same resource. These processes proceed to race attempting to acquire the resource. One process wins and acquires the resource. The others will have to go back to sleep again. On a monoprocessor, this is not a problem because the first to get scheduled receives the resource. By the time the rest of the processes are scheduled the resource is probably available to them. However, in a multiprocessor the awakened processes might all be scheduled simultaneously. There would be unnecessary context-switching, and context-switches have a tendency to invalidate the processors' caches. This would cause heavy system bus activity while the caches are being refreshed. In general, semaphores are more appropriate in a multiprocessor because they provide more structure and eliminate this type of unnecessary context-switching.

5.2 Interrupt Distribution

Perhaps the most important function of the SLIC is that of interrupt control and distribution. In order to eliminate the potential overloading of a single processor with the entire system's interrupt load, the SLIC supports dynamic interrupt distribution.

The SLIC subsystem handles all device interrupts in the system. Interrupts from peripherals on the Multibus* and the Small Computer Systems Interface (SCSI) bus are mapped to SLIC interrupt messages via Multibus-Adapters and SCSI host Adapters, respectively. All such SLIC interrupts are mapped to SLIC bins 1-6. Each bin has a potential for 256 interrupt vectors (specified by unique SLIC messages). Although this provides good flexibility, it is undesirable to build an interrupt vector table for all possible vectors. SLIC interrupts are programmed interrupts, so each module (processor, controller, etc.) on the system bus is told at system initialization time which SLIC bin and message data to send for each interrupt. Interrupt vector tables can then be optimally sized at system initialization time.

Each SLIC on the SLIC bus responds to interrupts directed at its own SLIC number. In addition, each SLIC corresponding to a processor is programmed to respond to the same destination group number. All device interrupts are directed to this group number. When an interrupt is injected into the system, the SLICs in the processor group arbitrate among themselves to determine which accepts the interrupt. Once the interrupt is accepted, the bin on the accepting SLIC is masked until completion of the interrupt handler so that this SLIC will no longer arbitrate for other interrupts in that bin. The acceptance of an interrupt by one SLIC in a group does not inhibit another SLIC in the same group from accepting another interrupt from the same bin or another bin. This allows multiple device interrupts to be serviced simultaneously.

MULTIBUS is a trademark of Intel Corporation.

Directing interrupts to a group of processors has several advantages. First, interrupts will be dynamically distributed among the group of processors. Thus no single processor or statically defined group of processors is a bottleneck. Secondly, processors may be brought online or taken offline dynamically without the loss of any interrupts. Finally, this architecture allows improved interrupt latency over that of a monoprocessor.

A SLIC arbitrates for interrupts based on its local priority register. The lower the priority, the more likely the SLIC will win the arbitration. The kernel sets the local priority register to reflect the scheduling priority of the process currently running on the processor. Idle processors set the SLIC priority register to the lowest possible value. The idea is to have the processors running the least important processes handle most of the interrupt load. Thus interrupt load is automatically and dynamically off-loaded from the processors doing the most important work.

5.3 Monoprocessor Device Driver Support

In order to support a monoprocessor-based device driver in a multiprocessor environment, it is necessary to simulate a monoprocessor environment for the driver. The key requirement is to have the monoprocessor's interrupt routine execute on the same processor as the driver's process-level code. If this requirement is met, the SPL mechanism for synchronization will be sufficient. The SLIC supports this requirement through the use of directed interrupts to an individual processor rather than to the normal processor group. There is no restriction on other drivers in the system and their interrupts will continue to be directed to the processor group and therefore distributed.

DYNIX supports monoprocessor drivers by:

- Initializing the device interrupt so that it is directed to a single processor M (currently the first processor booted.)
- Emulating the sleep()/wakeup() model. This is implemented via a hashed array of semaphores, where the hashing function translates the wait channel address to a semaphore.
- Binding process-level driver code to run on processor M. A process using the monoprocessor driver is context-switched to processor M and is bound to processor M while executing the monoprocessor driver code.
- Binding timeout routines to processor M, if a monoprocessor driver is configured. This ensures that a monoprocessor driver's timeout routines are properly synchronized with the rest of the driver.

5.4 Inter-processor Signaling

Inter-processor signaling is another multiprocessor issue solved via the SLIC. A processor may need to signal another processor when there is some task for the other processor to perform. An example is a preemptive rescheduling nudge. This signaling is implemented via programmed interrupts through the SLIC. With the SLIC, one processor may send another a normal maskable interrupt (Bin 1-7), a non-maskable interrupt, or a software interrupt (Bin 0).

Software interrupts are most commonly used for interprocessor signaling during normal operations. The main reason is that the sender does not have to wait for the destination to acknowledge the interrupt, as Bin 0 interrupts are always accepted by the destination SLIC. The software ties each of the 8 bits in the software interrupt message data to a particular software interrupt handler, the most common being the rescheduling nudge. In addition, non-maskable interrupts are sent to all processor SLICs when the system panics and needs to shut down rapidly.

5.5 Process Scheduling

The process scheduling technique in a symmetric multiprocessor is conceptually similar to that of a monoprocessor. Whereas the monoprocessor scheduling policy is to always execute the highest priority runnable process, the multiprocessor scheduling policy is to always execute the N highest-priority runnable processes, where N is the number of processors. The multiprocessor dispatching model must:

- Provide for dynamic load balancing.
- Dynamically adapt to N processors, for N >= 1.
- Avoid unnecessary process migration and context-switching, as each dispatch causes heavy bus activity until the processor memory cache is refreshed.
- Allow for the dynamic starting and stopping of processors.

The DYNIX scheduler uses a single priority-ordered queue of runnable processes **(RunQueue).** There is no processor-specific distinction made for processes in the RunQueue and there is no static binding of processes to processors. Since all processors are identical, any process (whether in User or Supervisor mode) may run on any processor. The symmetric, shared-memory architecture allows for easy implementation of dynamic load balancing. Since all processors are identical and all process state information, code, and data reside in a common shared memory, process migration is trivial.

The basic dispatching algorithm is simple. Each processor, upon entering the dispatcher, merely removes the highest priority process from the RunQueue and executes it. Access to the RunQueue is synchronized via a single SLIC gate. If there is no work to do, the processor executes its idle loop. The idle loop spins testing the RunQueue for runnable processes. Note that this testing does not require any SLIC gate synchronization, since it is sufficient to merely detect change in the RunQueue status. Once change is detected, the idle loop returns to the dispatching loop. The idle loop takes advantage of the bus-watching cache, and does not consume any system bus cycles or SLIC bus cycles. A simplified model of the basic dispatcher is illustrated in Figure 5.

```
swtch() {
        acquire RunQueue gate;
        while( RunQueue empty ) {
                release RunQueue gate;
                idle();
                acquire RunQueue gate;
        }
        remove highest-priority process;
        release RunQueue gate;
        set SLICPRI to that of the process;
        resume( process );
}

idle() {
        set SLICPRI to IDLE;
        while( RunQueue empty )
                continue;
}
```

Figure 5: Basic Dispatcher

Besides rescheduling themselves voluntarily, processes may be preempted either by a time-slice or when a higher-priority process becomes runnable. A time-slicing interrupt is injected into the system periodically to cause the accepting processor to determine whether there is a running process which should be preempted by a higher- or equal-priority process in the RunQueue. If so, the processor to be preempted is nudged to reschedule via a directed software interrupt. A process often becomes runnable as a result of a device interrupt. The processor accepting the device interrupt determines whether the awakened process should preempt a currently running process. If so, it will nudge the processor running the lowest priority process, telling it to reschedule.

The priority of the process running on a given processor is stored into the SLIC's priority register. This is used by the SLIC for contention resolution when arbitrating to receive an interrupt request. This supports a heuristic that the lowest-priority processor is most likely to handle an interrupt that causes it to be rescheduled. Setting the per-processor redispatch flag **(runrun)** may be the only action necessary to facilitate a reschedule if the interrupt makes a higher priority process runnable. This lowers the overhead in that the more expensive directed software interrupt is avoided and no additional SLIC bus cycles are needed to signal the context-switch.

5.6 System Configuration Control

In addition to its role as an interrupt controller, the SLIC subsystem provides a convenient, simple and reliable communication path among modules (processors, controllers, bus adapters, etc). This communication path is used to determine system configuration, to configure and deconfigure modules, to bring processors online or take them offline, and to exchange error management information.

The support for these diverse functions is obtained via SLIC slave registers. Each SLIC has the capability of either reading or writing any other SLIC's slave registers via SLIC messages. In addition to reporting status, SLIC slave registers also serve as command registers for their respective modules. The functions attached to these registers are module-dependent. For example, slave registers on the memory controller are used to report ECC error information, to identify the memory configuration (64K or 256K chip technology), and to set various configuration attributes of the controller (e.g., base address, interleave factor, etc). Also, processors are brought online and taken offline via remote SLIC messages directed at the desired processors' SLICs.

To facilitate system configuration, there is a subset of SLIC slave registers common to all modules. These include configuration information such as module type (e.g., processor, memory, etc) and revision level. With the SLIC slave register mechanism available to communicate configuration information and to set configuration attributes, there is no need for switches or wire-wrap stakes on the modules on the system bus.

The system's power-up firmware takes advantage of the SLIC bus to probe for the presence of hardware modules. It then builds a complete autoconfiguration table in a known place which can be used by diagnostics, the DYNIX operating system, or any other stand-alone program. When another processor board is plugged into the backplane, the power-up firmware will automatically add this resource to the configuration table. No changes to the software are necessary, and a single DYNIX binary is able to control a wide variety of hardware configurations. This simplifies the task of field upgrades and maintenance.

6. Evaluation

The Sequent Balance 8000 is an implementation of a shared-memory symmetric multiprocessor, currently supporting from two to twelve NSC Series 32000 microprocessors as compute engines. Each processor is packaged with a main-memory cache and a SLIC, allowing the operating system to take advantage of these features. The remainder of this section discusses some of the particular benefits derived and a few problems that result.

6.1 SLIC-Supported Symmetry

A major benefit of the per-processor SLIC and cache, and use of SLIC in all major hardware components of the system, is the resulting symmetry of the hardware as viewed from the software. The primary advantage is that all processors are literally *identical* in hardware and are so treated by the operating system. This is manifested in a number of ways:

- Any processor can service any interrupt at any time. This automatically distributes the interrupt load, using idle processors when they exist. This also allows processors to be dynamically removed from service, or taken *offline,* without impact on system operation.
- Any processor can control any piece of hardware, through the SLIC slave registers. In particular, any processor can take another offline or place it back online, or interrogate accumulated soft-error status in a memory controller.
- Any processor can signal another using SLIC interrupt messages. For example, there is no central dispatching authority since any processor can initiate a context switch in another. SLIC messages are often used to start I/O operations, avoiding any special affinity between processors and I/O-controllers.
- Any processor can be the first booted, eliminating the need for special treatment during system initialization.

The result is that no processor is special for any purpose. This allows any process or operating system code to execute on any processor at any time, greatly enhancing the load-balancing aspects of the system. The operating system treats the processors as just another resource to be managed, by arranging that processors dispatch themselves in response to process state changes.

6.2 Bus-Watching Cache

The operating system takes significant advantage of the bus-watching support in the cache. This simplifies low-level mutual exclusion and dispatching primitives without impacting overall system performance.

6.3 Few Hard-Coded Limits

There are very few hard-coded limits on the amount of hardware resource the system will support. There is no notion of the maximum number of processors that may be present in the system, although with enough processors some the low-level dispatching algorithms may be non-optimal. Similarly, the system can support an almost arbitrary amount of physical memory and number of I/O bus adapters, subject to hardware limits. Independence of these parameters allows a single operating system binary to run on a large number of different hardware configurations. In particular, the system may be shut down and one or more processor boards (or memory boards, I/O controllers, etc) added or removed; when the same kernel binary is rebooted the operating system automatically uses the new resources or adapts to the lack of a resource previously available.

6.4 Some Performance Data

Since the processors in a tightly-coupled multiprocessor must contend for the bus and memory, it is reasonable to expect that a system containing N processors will provide less than N times the performance of a single processor. In fact, a rule of thumb commonly applied to tightly-coupled multiprocessors predicts that each additional processor will provide at most 80% of the computing power of the previously added processor. Given this rule of thumb, system performance improvement would top out below 5 effective processors no matter how many are added.

The Sequent Balance 8000 system design refutes this rule of thumb. The combination of per-processor cache, SLIC, bus, and memory controller design provides a nearly linear increase in performance as the number of processors is increased from two to twelve, depending on the

application. Compute-bound applications benefit most from the addition of processors, and are the ones that see the nearly linear improvement. Sequent has run a number of benchmark programs in single-stream and multi-stream mode to measure the effective throughput of the system as the number of processes and processors are increased. Some of the programs intentionally try to break the cache, to measure a worst case situation. Figure 6 briefly presents the results; more detailed information is available from Sequent.

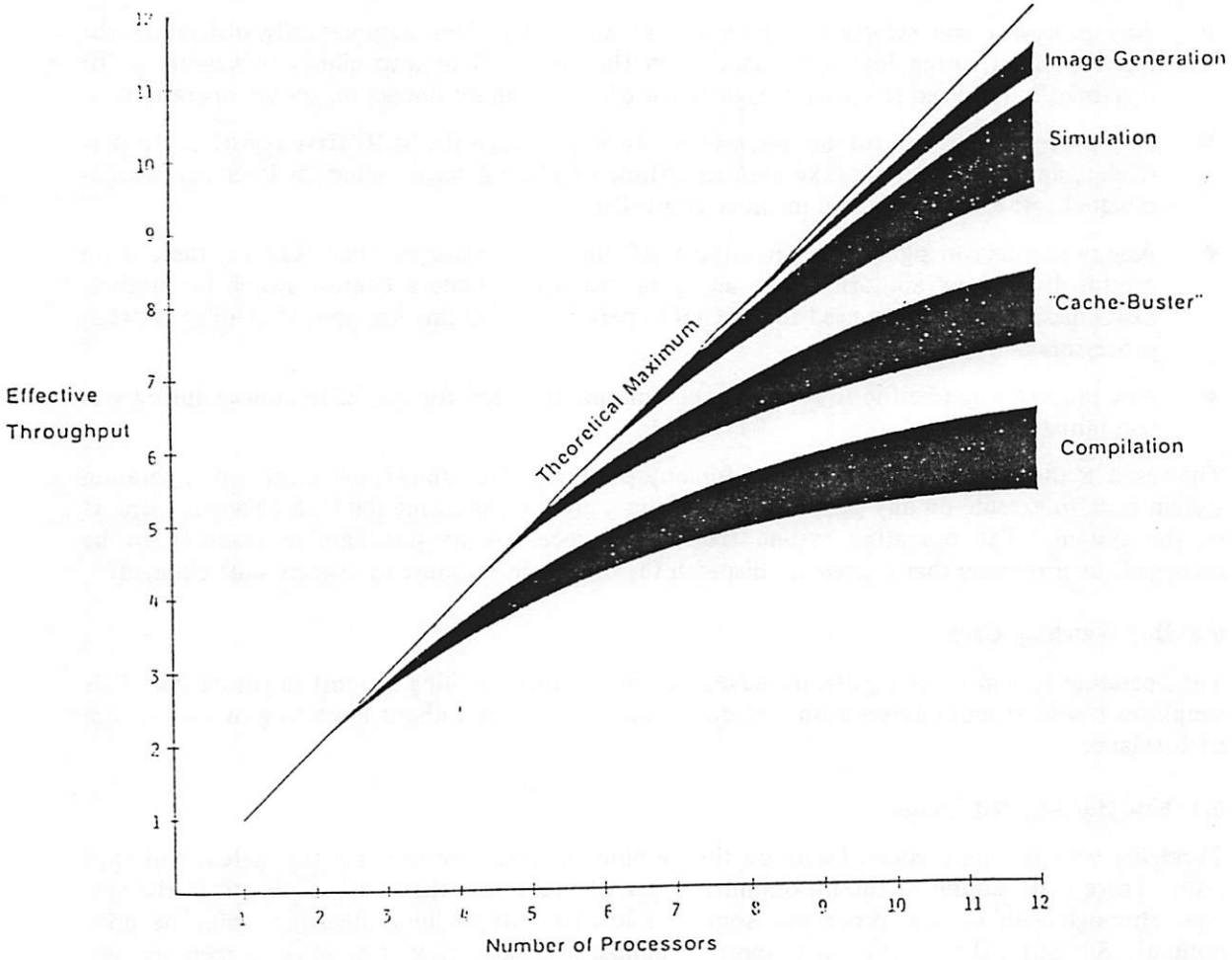


Figure 6: Effective Throughput vs Number of Processors

6.5 Gates vs Memory Test-and-Set

The decision to use SLIC gates instead of test-and-set memory support was based on a desire to keep the system bus, memory controllers, and processor cache implementation simpler and lower cost, while achieving good performance. Separating the SLIC bus from the main system bus allows both to be simpler and more tuned to their respective purpose. Since 64 gates are not sufficient for all low-level data structure mutual exclusion, the *lock* abstraction of gates was created. The actual collision rate on the gates distributed among locks and semaphores is very low, typically less than 1% of all gate transactions. There are a few gates that exhibit higher collision rate; in particular, the RunQueue gate and timer-structures gate. But even in a system that supported memory-based test-and-set operations, these data structures would each be controlled by a single test-and-set variable, so using gates creates no additional problem.

A problem with the limited number of gates is managing them as a resource. Currently they are statically allocated, defined mnemonicly in a header file, so the entire system must be recompiled to change the assignment of gates to functions. There also must be a few gates reserved for use by relatively orthogonal parts of the operating system (such as device drivers). Fortunately, the initial distribution of gates seems to work well. This is an area that needs further study.

6.6 User Access To SLIC

SLIC is so central to correct operation that incorrect use can easily crash or deadlock the system. As a result, direct access to the SLIC by user processes is strictly forbidden. The motivation is similar to disallowing direct user access to important kernel data structures. Applications that need access to the SLIC (e.g., to send interrupt messages or access slave registers) typically create a device driver or other code that lives within the operating system for this purpose (all kernel code may access SLIC).

One result of this is that user processes cannot access gates other than by calling the operating system. Applications that want to take explicit advantage of multiple processors running in parallel in a shared memory usually require fast mutual exclusion support, much like the operating system. Calling the operating system for this purpose takes one or two orders of magnitude more time than can be tolerated by many such applications. There are various software solutions to this problem [Dijkstra 1965, Lamport 1974], but these are awkward to use and don't provide optimum performance. To address this need the Multibus Adapter supports a set of test-and-set variables, called **atomic lock memory**. The Multibus adapter can be mapped into user process address space, to provide convenient and fast access to these variables. Since the bus treats I/O accesses separately from memory accesses, manipulation of atomic lock memory variables does not impact system bus bandwidth.

7. Summary

Microprocessors have reached a point of maturity where multiprocessor systems can be built using microprocessors as the computing engines, leveraging VLSI technology to reduce cost, package size and complexity, and increase reliability. It is appropriate that additional VLSI leverage can successfully be applied to certain other problem areas:

Cache Memory. Symmetric, tightly-coupled multiprocessor systems place a high demand on system bus and memory bandwidth. Properly coordinated per-processor cache memories significantly reduce bus traffic and memory contention, avoiding traditional bottlenecks.

Interrupt Distribution. An intelligent interrupt controller can support dynamic distribution of interrupt load among processors. The advantages include:

- Automatic interrupt distribution on a process priority basis.
- Enhanced system availability by not binding interrupt sources to particular servers.
- Improved interrupt latency.
- Transparent assist to dynamic load balancing.
- Integrated inter-processor signaling mechanism.
- Software-assignable interrupt vectors, enhancing configurability.
- Simple accommodation of monoprocessor drivers.

Inter-processor Synchronization. An efficient low-level mutual exclusion primitive can be provided inexpensively, and can support the implementation of classic higher-level synchronization primitives.

System Configuration and Control. Use of a common mechanism to identify and control basic hardware components in the system allows one operating system binary to run on a wide variety

of hardware configurations. This reduces the complexity of hardware and software configuration, and simplifies maintenance tasks. System resources can be brought in and out of service dynamically, and error information can be gathered, all using the same mechanism.

8. Conclusions

Shared-memory symmetric multiprocessor systems have long been of interest due to their potential for high performance and flexibility. In the past, barriers such as entry cost, achievable range of performance, and ease of use have limited their commercial success. Multiprocessors were once thought to be limited to a performance level of 3 to 4 times that of a monoprocessor. However, the system we have described realizes an almost linear improvement in performance as processors are added. By exploiting VLSI technology, low-cost, high-performance multiprocessors can be built which fully support UNIX.

References

[Dijkstra 1965]
Dijkstra, E. W., "Solution of a Problem in Concurrent Programming Control", Communications of the ACM, Vol. 8 No. 9, September 1965.

[Efland 1984]
Efland, Greg, and Walter Mayberry, "Cache Boosts Multiprocessor Performance", Computer Design, November 1984.

[Fielland 1984]
Fielland, Gary, and Dave Rogers, "32-Bit Computer System Shares Load Equally Among Up To 12 Processors", Electronic Design, September 6, 1984.

[Gobel 1981]
Gobel, George H., and Michael H. Marsh, "A Dual processor VAX 11/780", School of Electrical Engineering, Purdue University, TR-EE 81-31, September 1981.

[Goodman 1983]
Goodman, James R., "Using Cache Memory to Reduce Processor-Memory Traffic", Proceedings of the 10th International Symposium on Computer Architecture, pp. 124-131, June 1983.

[Lamport 1974]
Lamport, L., "A New Solution of Dijkstra's Concurrent Programming Problem", Communications of the ACM, Vol. 17 No. 8, November 1974.

[Lovett 1984]
Lovett, Tom, "A CAE Case History", VLSI Design, November 1984.

Implementing Loosely Coupled Functions on Tightly Coupled Engines.

Jack Inman

Sequent Computer Systems
14360 NW Science Park Drive
Portland, Oregon 97229

ABSTRACT

This paper describes the experiences of porting UNIX[1] 4.2bsd networking functions to DYNIX,[2] a tightly coupled, multiprocessor implementation of UNIX 4.2bsd. The paper identifies networking issues in a multiprocessor system and generalizes multiprocessor system concepts by way of relating examples of them encountered in the Interprocess Communications (IPC) portions of the concurrently executing DYNIX kernel. It first briefly describes the UNIX 4.2bsd IPC model, then the DYNIX multiprocessor model and how it is applied to IPC. It further discusses some of the things that can go wrong, how these situations were discovered and corrected, and finally gives examples of the general issues and effective implementations.

1. Introduction

The author is responsible for the IPC portions of Sequent's DYNIX kernel. DYNIX is a kernel based on UNIX 4.2bsd that executes concurrently on multiple processors. The development of DYNIX IPC involved "porting" the UNIX 4.2 IPC architecture and algorithms to a multiprocessor system. The nature of the development, i.e. it being a port of existing architecture and algorithms, offers the benefits of many algorithmic solutions already implemented in UNIX 4.2bsd. Furthermore, porting suggests the ability to quickly absorb other protocol implementations under UNIX 4.2bsd as they are developed. A faithful port also offers a good degree of assurance that the semantics of the protocols from the points of view of both the user and the network are maintained. The drawbacks to the approach however are that 1) concurrent programming is not an easy thing to do anyway, 2) the base implementation executes in a monoprocessor system and thereby dictates certain semantics, and 3) characteristics of networking systems introduce additional multiprocessor pitfalls. It requires continuous judgement on when to port an algorithm, a set of semantics, or a particular data structure, and when to redesign or enhance the design to accommodate concurrent processing. DYNIX is evidence of the success of this effort and this paper shares some of the lessons learned.

1 UNIX is a trademark of AT&T Bell Laboratories.

2 DYNIX is a trademark of Sequent Computer Systems, Inc.

2. Rationale for the DYNIX IPC Model

A good question to answer at this point is why do this? Since multiple processors are available in the DYNIX host, and UNIX 4.2bsd provides useful protocol implementations implemented within the kernel, it is advantageous to implement IPC services within the DYNIX kernel to execute in a multiprocessor host. Furthermore, with a general multiprocessor model, greater advantage can be taken of concurrency as the number of processors increases. This means for example that as networking needs grow, more processors can be effectively added.

There are alternative approaches. For example, a loosely coupled approach, in which the IPC portions of the kernel are moved to intelligent controllers is one alternative. This approach is popular in monoprocessor environments because it off loads the single host processor of a lot of processing requirements. However, in general it does not cover all interesting cases since significant IPC services (notably pipes) must be executed in the hosts. Furthermore, such loosely coupled implementations have disadvantages in internetworking configurations.

A different IPC model could have been chosen such as "transparent" distributed file system, but this suffers from nonstandardization. Such distributed file systems typically work well only with themselves. Interoperability and compatibility with existing equipment are key objectives of DYNIX.

There are benefits of implementing the IPC services within DYNIX. The major benefit of course is performance. This is not to say that a stream of bytes can be transferred from point A to point B quicker using DYNIX than a monoprocessor implementation of the same protocols. Consider, the sequential algorithms are essentially equivalent, being ported from a monoprocessor implementation. Furthermore the class of processor (National Semiconductor's Series 32000) is essentially equivalent to a VAX 750[3] in silicon. It is difficult to imagine performing any better than a comparable monoprocessor implementation with respect to how long a particular network transaction takes end to end. The issues are concurrency and total system performance. I.e. how much total work gets completed by the system including, or especially, network related work.

Significantly, concurrency allows user processes to make progress while network services are being executed. For example, user response time is less affected by network activity since a separate processor can do the network chores. This generally results in the system being able to make more IPC service requests, thereby making more work available more often. Concurrency further allows different pieces of the network services to be executed by multiple processors concurrently, thereby improving total throughput due to better service to the network.

2.1 Concurrency and "Layered" Communications Model

Contemporary communications subsystems are "layered". They are often modeled after the ISO Open Systems Interconnect Architecture [Zimmerman 1980]. There are many benefits to this layering and examples of its effectiveness.

It is natural to imagine the application of multiple, concurrently executing processors *vertically* through this model. For example, a processor might be allocated per protocol layer, or as often the case in practice, several layers might be grouped together in "macro" layers. Vertical concurrency is exemplified by several multiprocessor architectures, including loosely coupled architectures such as various intelligent front end network controller models. Although a degree of concurrency is achieved with these models, it is available only for network services and only applicable vertically through the ISO layers. In affect, the processor on the controller is not available for anything else. With the cost of microprocessors decaying so rapidly, this is acceptable idle time in some situations, but having the ability to perform the network function within a general purpose computer environment has many advantages.

---

3 VAX is a trademark of Digital Equipment Corporation

For example consider the effect of three processors allocated in a well balanced fashion to perform a network function such as a file transfer. One processor is busy performing user process tasks such as obtaining the data and communicating with the user. Another processor is busied managing network tasks such as packetizing the data, checksumming, and executing various protocol management algorithms. It is doing this for several streams of network data flow. The third processor is handling the low level interface to the transmission medium. This scenario creates an effective pipeline of processors doing a lot of work without having to context switch. The movement of any one piece of information is somewhat more expensive than it would be with only one processor executing the tasks, but the total amount of data that is transferred is more. In practice, of course, the situations are much more dynamic. In the long run, fair, dynamic allocation of processors to concurrently executing tasks improves total system throughput. DYNIX IPC accomplishes such concurrency with many more degrees of freedom than this simple example. The actual hardware on which DYNIX executes, the Balance 8000,[4] supports up to twelve NSC Series 32000 microprocessors. Several network functions execute concurrently as well. The implementation accommodates 1 to 12 processors and benefits from each incremental number of processors.

Using a tightly coupled multiprocessor model a dedicated controller processor is not required for network processing. This lowers the cost of the connection. For example, in the case of a Balance 8000, an interface to the Ethernet[5] does not require a controller per se. The Ethernet data link is simply one function of a multi-purpose controller board [Sequent 1984].

Horizontal concurrency is also possible. An obvious consideration in a multi-user and multiprocess environment such as UNIX is concurrent execution of user application protocols. Multiple communications functions such as file transfer and virtual terminal can execute on several processors sharing the same operating system services. Gateway servers might take advantage of multiple concurrently executing network interfaces. It is also possible for multiple concurrently executing protocol suites, such as XNS[6] and TCP/IP to operate over a single Ethernet data link. Transmit and receive functions can also execute concurrently, generally (but not always) independent of each other. Other concurrent execution possible includes timer events and network management functions such as routing table updates.

2.2 UNIX 4.2bsd Compatibility as a Goal

A significant objective of this port is to be fully compatible with UNIX 4.2bsd both from the user interface and perhaps more importantly, from the network interface. A user program that makes use of the UNIX 4.2bsd IPC interfaces should be readily portable to a Balance 8000 system. Existing (and future) applications of IPC services should also work on current (and future) DYNIX systems. Examples of such user applications are the "r commands" which UNIX 4.2bsd supports, viz. *rlogin, rsh* and *rcp*. From the Unix Programmer's Manual pages [UNIX 1983]:

rlogin (1C)	- remote login
rsh (1C)	- remote shell
rcp (1C)	- remote file copy

These applications are supported in DYNIX with virtually no changes other than recompilation. Furthermore, since compatibility is also achieved at the network interface, these applications function well across a network to other UNIX 4.2bsd implementations. That is to say, a Balance 8000 hosting DYNIX easily communicates with a VAX hosting UNIX 4.2bsd and vice versa. One can *rlogin, rsh,* or *rcp* files to and fro.

4 Balance is a trademark of Sequent Computer Systems, Inc.

5 Ethernet is a trademark of Xerox Corporation.

6 XNS is Xerox's Networking Systems protocol suite

For example, the Sequent company environment contains several different kinds of computer systems including multiprocessor DYNIX systems and monoprocessor UNIX 4.2bsd systems. Using these applications, all that is needed is a nickname for the machine. Since the interfaces are the same, it is easy to use them. Once logged in, it is difficult to tell a monoprocessor UNIX 4.2bsd machine from a multiprocessor DYNIX machine (except for the response time and load average statistics). In the Sequent company environment most people in the company have reason to use resources on several different machines. This includes people that do not have development responsibilities. Interesting evidence to support the conclusion that most users access multiple hosts lies in the fact that one of the most used commands on the production machines is *hostname*[7] which is often placed in user's prompts to remind them where they are. The compatibility is sufficient to allow one to forget what type of machine it is that they are talking to (until they try to execute a NSC Series 32000 binary file on a VAX).

2.3 Network Interoperability

A major rationale for porting the supported Internet protocols is the degree of interoperability which standards can achieve. Balance 8000 can immediately participate on networks of computers executing UNIX 4.2bsd, but also on many networks with computers that support implementations of the Internet protocols over the Ethernet. These include the DARPA[8] functions *telnet* (comparable to *rlogin*) and *ftp* (comparable to *rcp*). From the UNIX Programmer's Manual pages [UNIX 1983]:

ftp (1C) - file transfer program
tftp (8C) - DARPA Trivial File Transfer Protocol
telnet (1C) - user interface to the TELNET protocol

Using these applications, DYNIX can also communicate through existing gateways to other implementations. It is a significant win for a new product to be able to "talk" to existing solutions without having to develop special hardware or software. Balance 8000 and DYNIX achieve this goal.

3. The UNIX 4.2bsd IPC Model

The following briefly describes details of the UNIX 4.2bsd Communications Architecture. In this architecture IPC services are integrated into the kernel. Consequently IPC services are also integrated into the concurrent DYNIX kernel. It is assumed that the reader is somewhat familiar with the UNIX 4.2bsd model such that brief context is sufficient and is referred to the references for further insight [Leffler 1983, Leffler 1983B]. Also, since DYNIX implements the same general architecture with the enhancement of concurrent execution, the terms UNIX 4.2bsd, DYNIX, and kernel are used somewhat interchangeably.

With respect to IPC the kernel provides functions that:

[1] Define and manage data structures to support the communications architecture.

[2] Initialize the communications system.

[3] Provide the system interfaces that map user requests to communications services.

7 hostname (1) - set or print name of current host system [UNIX 83].

8 DARPA is the Defense Advanced Research Projects Agency network which defines a set of standards for computer to computer communications.

[4] Provide the system interfaces that map communications services to user requests.

[5] Manage the kernel's communications system buffers *(mbuf*'s).

[6] Manage the movement of data from kernel space, to user space.

[7] Provide timer events.

At the base of the communications definition is the notion of communications **domain**. [Leffler 1983B] defines a communication domain as "an abstraction introduced to bundle common properties of processes communicating through sockets. One such property is the scheme used to name sockets."

In the kernel the domain structure consists of a linked list of the domain entries supported by the system. Consequently, multiple domains are supported by the architecture. The domain entries are used to find the appropriate **protocol switch**, *struct protosw*. Each domain includes an array, *protosw*[], of *protosw* entries. A single domain supports multiple protocols each one represented by a *protosw* entry.

An **Address Family** specifies a communications domain. For example, the Internet domain is identified by the value AF_INET. There are two communications domains currently supported in DYNIX:

[1] UNIX domain (AF_UNIX),

[2] Internet domain (AF_INET).

The AF_UNIX domain is used for UNIX to UNIX local interprocess communication. It for example includes communication via UNIX pipes. The AF_INET domain is used for both local and remote interprocess communication using the DARPA protocols.

There are three types of communications available represented by **socket types**. These are *stream* (SOCK_STREAM), *datagram* (SOCK_DGRAM), and *raw* (SOCK_RAW) socket types.

AF_UNIX supports three *protosw* entries, one for a SOCK_STREAM, one for a SOCK_DGRAM, and one for a SOCK_RAW socket interface. Pipes are included in the SOCK_STREAM definition. AF_INET has five *protosw* entries corresponding to the Internet Protocol (IP), Internet Control Message Protocol (ICMP), User Datagram Protocol (UDP), Transmission Control Protocol (TCP), and a "raw" internet protocol (rip_). Not all of the information is required for every *protosw* entry. Entries are NULL if not needed.

At system initialization, *domaininit()* is executed to link the domain data structures and call the protocol initialization procedures via the *protosw*[] entries. In this way the domain structure is used to define a protocol family's features.

For purposes of further discussion the IPC services are divided into Protocol Engines, Socket Management, and Network Interface Management.

3.1 Protocol Engines

This paper uses the notion of **protocol engines** to represent the protocol specific functions executed by the kernel. This includes the algorithms and data structures used to implement a particular set of protocols. Protocol engines are defined in part by the routines specified in a domain's *protosw*[]. They perform the functions of the protocol including transmission and delivery of data, and control of that transmission and delivery. A *protosw* entry is defined [Leffler 1983]:

```
struct protosw {
  short                         pr_type;          /* socket type */
  short                         pr_family;        /* protocol family */
  short                         pr_protocol;      /* protocol number */
  short                         pr_flags;         /* see below */
      /* protocol engine handles */
  int                           (*pr_input)();    /* input (from below) */
  int                           (*pr_output)();   /* output (from above) */
  int                           (*pr_ctlinput)(); /* control input (from below) */
  int                           (*pr_ctloutput)(); /* control output (from above) */
      /* user-protocol hook */
  int                           (*pr_usrreq)();   /* user request: list below */
      /* utility hooks */
  int                           (*pr_init)();     /* initialization hook */
  int                           (*pr_fasttimo)(); /* fast timeout (200ms) */
  int                           (*pr_slowtimo)(); /* slow timeout (500ms) */
  int                           (*pr_drain)();    /* flush excess space */
};
```

Socket management and protocol engines routines pass commands to the protocol engine via the protocol engine handles. The commands passed to the protocol engines include:

```
#define PRU_ATTACH      /* attach protocol to up */
#define PRU_DETACH      /* detach protocol from up */
#define PRU_BIND        /* bind socket to address */
#define PRU_LISTEN      /* listen for connection */
#define PRU_CONNECT     /* establish peer connection */
#define PRU_ACCEPT      /* accept peer connection */
#define PRU_DISCONNECT  /* disconnect from peer */
#define PRU_SHUTDOWN    /* won't send any more data */
#define PRU_RCVD        /* data taken; more room now */
#define PRU_SEND        /* send this data */
#define PRU_ABORT       /* abort */
#define PRU_CONTROL     /* control ops on protocol */
#define PRU_SENSE       /* return status into m */
#define PRU_RCVOOB      /* retrieve out of band data */
#define PRU_SENDOOB     /* send out of band data */
#define PRU_SOCKADDR    /* fetch socket's address */
#define PRU_PEERADDR    /* fetch peer's address */
#define PRU_CONNECT2    /* connect two sockets */
#define PRU_FASTTIMO    /* 200ms timeout */
#define PRU_SLOWTIMO    /* 500ms timeout */
#define PRU_PROTORCV    /* receive from below */
#define PRU_PROTOSEND   /* send to below */
```

These routines are the kernel's handles into a protocol engine. Socket management uses the appropriate *pr_usrreq()* handle to service user process requests (requests from "above") and network processes use them to access protocol engines from "below". They are also used from within to execute initialization and timer events.

A protocol engine is exemplified by the Internet protocol engine for the Internet protocol set. It supports the DARPA standard Transmission Control Protocol (TCP), User Datagram Protocol

(UDP), Internet Control Message Protocol (ICMP), and the Internet Protocol (IP).

Note, that the Communications Architecture is independent of actual protocols employed, and it accommodates multiple communications protocols within the same system. Although there is a defined set of protocol engines currently supported by DYNIX, the architecture supports other protocol engines. There is further analysis that proposes some minor improvements to make the architecture even more general. [O'Toole 1985].

3.2 Socket Management

A key abstraction in the communications architecture is that of a **socket**. A socket is what a user process uses to access IPC services, such as send and receive. Socket management provides the binding of user processes to sockets. It therefore manages movement of data in and out of user processes and the kernel, creation and deletion of IPC resources, and reference to protocol engine services (i.e. the execution of the appropriate communications protocols).

A socket data structure contains pointers to the protocol control information, input and output data queues, socket state variables, and other variables used by the system. It is used by the kernel to process user requests, network requests, and time out requests. It is created as a result of a *socket()* UNIX 4.2bsd system call.

A program references a socket in order to affect interprocess communication. A socket is analogous to a file descriptor. In fact, internally sockets are defined as a type of file descriptor. Consequently, kernel file system services are used by the IPC implementation to help manage socket data structures.

For example, when a *socket()* system call is made, a file descriptor is acquired from the user process' available set of file descriptor resources. These are managed by the kernel just like any other file descriptor. When the socket abstraction is closed, either explicitly, or implicitly due to process termination, the kernel uses the file descriptor to initiate deallocation of IPC resources.

The IPC system calls available to a user process are described in the UNIX Programmer's Manual pages [UNIX 1983]:

read, readv (2)	- read input (from SOCK_STREAM)
write, writev (2)	- write on a file (to SOCK_STREAM)
close (2)	- delete a descriptor (E.g a socket)
pipe (2)	- create an IPC channel
ioctl (2)	- control device (or net parameter)
gethostname, sethostname (2)	- get/set name of current host
select (2)	- synchronous i/o multiplexing
socket (2)	- create an endpoint for IPC
connect (2)	- initiate a connection on a socket
accept (2)	- accept a connection on a socket
send, sendto, sendmsg (2)	- send a message from a socket
recv, recvfrom, recvmsg (2)	- receive a message from a socket
bind (2)	- bind a name to a socket
getsockopt, setsockopt (2)	- get and set options on sockets
listen (2)	- listen for connections on a socket
shutdown (2)	- shut down part of a FDX connection
socketpair (2)	- create a pair of connected sockets
getpeername (2)	- get name of connected peer
gethostid, sethostid (2)	- get/set unique id of current host
getsockname (2)	- get socket name

3.3 Network Interface Management

Protocol engines and socket management provide data to and receive data from any of several physical networks via network interfaces. Network interfaces define drivers provided for transmitting and receiving data over a network's physical media. The network interfaces control the hardware which does data transfer. Data passes through the network interfaces via input and output queues defined in network interface structures, *struct ifqueue* in *struct ifnet*.

Network interface output queues are typically serviced by transmit complete interrupt handling routines. Protocol engine routines queue output requests to the interface to be transmitted asynchronously. Received data is placed into input queues by receiver interrupt routines. These queues allow the protocol engines to execute at lower priority than hardware interrupt handling. On input, appropriate data link **clients** are started via **software interrupts** described below.

The most notable example of a hardware network interface is the Ethernet. Multiple Ethernet interfaces are supported in both UNIX 4.2bsd and DYNIX. Packets are received from an Ethernet interface and demultiplexed to clients by using the type field defined in the Ethernet standard [DEC 1982]. The most interesting data link client for discussion in this paper is the Internet protocol engine which receives Internet packets from one or several network interfaces. It processes the received data according to the appropriate protocol, E.g. TCP, UDP, or ICMP. It may also choose to perform internetwork routing and forward the data packet to another network interface.

UNIX 4.2bsd uses a mechanism referred to as a **software interrupt** to partition network functions. This allows lower level, but higher priority functions such as those done by the driver to execute quickly and schedule higher level functions such as the protocol engine functions for later execution at a lower priority. The data is queued into a network interface queue and when the situation permits, the higher level protocol engine executes. This facility provides good system response to the network interface when the network is busy.

4. The DYNIX Multiprocessor Model

DYNIX is a concurrently executing version of the UNIX 4.2bsd kernel. It provides three mutual exclusion (**mutex**) mechanisms, gates, locks and semaphores [Beck 1984]. The interfaces to these mechanisms include familiar "P and V" semantics from Dijkstra's *Proberen*and *Verhogen* ideas [Dijkstra 1965]. *p_xxx* attempts to acquire the resource, and *v_xxx* releases it.

Gates are used for low level, high priority processes such as memory management, low level structure management, and interface management. Critical sections protected by gates are kept to a minimum due to the fact that processors do not make progress while waiting for a gate to become available. Also there are a limited number of gates in the system supported by a proprietary silicon device, the System Link and Interrupt Controller (SLIC) [Fielland 1984]. A *p_gate()* results in a SLIC operation to acquire a gate. A *v_gate()* results in a SLIC operation to release it.

Locks are implemented using gates and are software managed mutual exclusion structures. *p_lock()* acquires the lock and *v_lock()* releases it. A processor can "spin" attempting to acquire a lock, but can be interrupted by a higher priority process if required. A processor execution priority level (SPL) is specified on the lock calls. However, keep in mind that SPL is insufficient for mutual exclusion in a multiprocessor system since other processors can be executing at the same, or even lower SPL concurrently [Bach 1984]. This is discussed further below.

The third mutual exclusion mechanism is the counting semaphore. A *p_sema()* acquires the semaphore if available and places the process on a queue if it is not available. A *v_sema()* starts the next process in line waiting for the semaphore. There is also a *vall_sema()* which starts all processes currently queued on the semaphore.

In addition to the standard "P & V" interfaces, DYNIX also provides "conditional" requests for locks and semaphores similar to those described in [Bach 1984], *cp_lock()* and *cp_sema()*. These interfaces are useful for cases where something else can be done rather than contend for the resource.

Another useful mutex interface is the *p_sema_v_lock()* call which allows a semaphore to be requested and a lock to be released in one call. This for example is useful to prevent semaphore structures from being deleted while in the process of being requested. It also helps to avoid "sleeping" while a lock is held.

4.1 DYNIX IPC Kernel Concurrency Model

The IPC portions of the kernel effectively make use of the DYNIX mutual exclusion mechanisms described to accommodate concurrency. The following describes their general use.

Semaphores are used to mutex user processes from each other, to signal when particular events occur such as connect or disconnect complete, and to contend for memory resources. A global semaphore is used to simplify some of the UNIX domain non-pipe IPC. Use of semaphores allows processes to wait for resources to become available, thereby allowing a processor to do something else until the resource is available. There are three semaphores defined for each socket. They are used for connection/disconnection signaling, and synchronization of requests for the input and output data queues.

Locks are used for lower level mutex generally involved with structures shared with protocol engines. For example, each socket structure has a lock associated with it. By assigning a lock per resource, unrelated user processes do not contend for the same resource for unrelated IPC. Locks are also defined for several protocol engine lists and the network interface queues.

Gates are used for the lowest levels of IPC services such as network buffer memory management, and low level network interface management. Gates are also used to mutex the software lock structures. IPC uses a total of three gates.

5. Interesting Networking Issues

Porting the IPC kernel to a multiprocessor system encounters many issues common to the general problems of implementing multiprocessor systems, but some issues are particularly germane for networking environments. Examples include network interrupts, unsolicited data and remote initiation of shared resource modification.

5.1 Network Interrupts

As described above, UNIX 4.2bsd implements software interrupts in order to provide good response to a busy network. DYNIX easily accommodates this feature and indeed, high level protocol engines execute concurrently with multiple network interface interrupt handlers. For input to a protocol client, data is placed into an input queue and a software interrupt is signaled. In a monoprocessor system the software interrupt is executed later at a lower priority. In a multiprocessor system it can execute immediately, or already be executing when the data arrives. Thus the first order of concurrent processing is easily achieved in DYNIX by properly managing the interface queues and software interrupt mechanism. This is relatively straight forward and uses a simple *p_lock()/v_lock()* protocol to enqueue and dequeue packets on the network interface.

This means that network interfaces can provide good service to network devices while protocol engine functions execute concurrently. The protocol engines avoid multiple instantiations of the software interrupt queue handlers, therefore, if the traffic warrants it, the processor continues to operate as a protocol engine. In a sense, the processor becomes a temporarily dedicated controller until something else more important requires the processor resource.

5.2 Unsolicited Data

An interesting characteristic of network subsystems is the frequent receipt of **unsolicited data.** This is described as "spontaneous input" in [Requa 1985] and is familiar to most developers of network or real time code. The issue is more pronounced in systems distributed on a Local Area Network (LAN) because typically packets of data arrive at random, usually bursty, intervals from multiple sources and potentially destined for multiple protocol engine clients. The receiving system must be ready to receive the packets or risk loosing them. Packet loss is not necessarily critical in networking subsystems since their purpose is to provide a level of deliverable service which assumes unreliable transfer. Nonetheless, it affects total system performance and missing packets is undesirable.

Typically, systems manage unsolicited data by buffering ahead. This can be as simple as two level buffering, where the network software immediately sets up the next input buffer upon receiving a block of network data. The more buffers available, the more back to back packets can be received without loosing them due to lack of buffers. However, it is impossible to anticipate how much data to expect for each packet since network data on a LAN consists of both large and small packets. Shoch's analysis shows Ethernet traffic for a comparable environment is bimodal [Schoch 1979]. This is true with TCP/IP also. In fact, the bimodalness of typical TCP/IP LAN traffic is so dramatic that "histograms" are formed simply by printing the decimal counts of the packet sizes seen. For example, the following is a picture of such analysis performed by monitoring Sequent's busy Ethernet under typical load:

```
total_packets -        27347608
packet counts per bucket ->
                    buck[0] = 15450762
                    buck[1] = 923132
                    buck[2] = 442860
                    buck[3] = 367880
                    buck[4] = 1052369
                    buck[5] = 9110664
```

The value of buck[0] represents the number of packets that contain under 64 bytes and buck[5] represents the number of packets that contain more than 1024 bytes. The decimal numbers themselves form a bimodal histogram. The buck[0], or short packet number represents 56% of the total packets and buck[5] or long packet number represents 33% of the total packets in this particular environment. The counts for [1] 64-128, [2] 128-256, [3] 256-512 are 2 decimal digits shorter than the counts for short packets and 1 decimal digit shorter than the counts for long packets. This is understandable considering protocol control packets and virtual terminal traffic are generally contained in short packets and file transfer generally attempts to use predominantly large packets.

This is of course just a snapshot of a particular period of network usage on a particular network and it changes with time and usage. For example, dumps across the network using the Unix 4.2bsd commands [UNIX 1983]:

```
rrestore (8C)    - restore file system dump across the network
rdump (8C)       - file system dump across the network
```

skew this more towards larger buffers.

The point is that for such unsolicited data, the system needs to be prepared for just about anything, at any time. Buffering ahead provides this preparation. DYNIX buffers ahead using chained 128-byte buffers. The number of these receive buffers is a configurable parameter that defaults to 200. The Ethernet driver attempts to keep this number of network buffers available for unsolicited input at all times. The more memory allocated to this function, the more packets can be queued during a burst of data. At the expense of tying up some system RAM, optimal

performance is ensured.

The replacement of network buffers does not always succeed and the network interface must be prepared to catch up at the next opportunity. This is accomplished by the network receiver interrupt handler requesting enough buffers to fill its pool. If a request for buffers is denied, it is picked up on the next receiver interrupt. The driver always keeps at least one buffer queued in order to keep things active, even if it has to discard the last packet received.

5.3 Remote Initiation of Resource Modification

A fundamental difference between tightly coupled systems and loosely couple systems is the degree of asynchronism. Loosely coupled systems are characterized by comparatively long delays between requests and responses. Consequently, a request is made by a system and it usually chooses to do something else for awhile rather than wait for the response. Such behavior means that an unsolicited input from a network can result in the creation, deletion or other modification to a shared resource. For example, consider what occurs with a *listen()/accept()* sequence. From the UNIX Programmer's Manual pages [UNIX 1983]:

listen (2) - listen for connections on a socket
accept (2) - accept a connection on a socket

These UNIX IPC services are used to establish an IPC server. The server uses a *listen()* system call to inform the system that it intends to receive connection requests from one or more clients. It uses a standard socket descriptor which it created to do this, however, the *accept()* system call returns a different socket descriptor. The listen socket is essentially used as a template for connection requests to describe the communications for subsequent *server/client* conversations. Each new request for connection from a potential client results in the creation of a new socket structure. The new socket is created asynchronously and initiated by a client process that can reside on a different host. When the connect request is received by the network, a new socket is created using the *sonewconn()* kernel function and queued to the server process using the listen socket. The new socket is subject to the same mutex considerations as sockets created via user requests and can also be aborted before they are completely established for protocol engine reasons. This is particularly interesting in a multiprocessor system since all of this occurs concurrently along with other processing as well.

Another example of remote initiation of resource modification is dynamic reconfiguration of network topology. Internetworking situations can result in dynamic modification of routing information or other connection state initiated by unsolicited network input. As an example, if a host or network suddenly becomes unreachable due to a system failure, Internet systems communicate with each other and direct that existing connections through failed systems be aborted. This too causes interesting dynamics in a multiprocessor system.

5.4 Concurrency Issues With Multiple Users and User Processes

The UNIX operating system supports the notion of many simultaneous users and each user can *fork()* a number of user processes executing on the user's behalf. The UNIX model permits, in fact encourages, processes to *fork()* children and allow them to share system resources that they initiated, for example standard input, standard out, and newly created server sockets. Typically, a parent process allocates the resource, then forks the child, then no longer accesses the resource. Typically, child processes do not share these resources although the possibility is available to them. However, typical execution cannot be the model, and since these resources are sharable, it is assumed that they will be shared, intentionally or otherwise. Therefore, once a socket is created, it can be accessed by one or more processors concurrently in the form of user processes and their child processes requests.

The monoprocessor kernel implements some consideration for user processes sharing a socket resource using busy and want flags on socket buffers and *sleep()/wakeup()* semantics. If multiple processes attempt to read from a socket, they are serviced on a whoever gets there first basis. Note this is not First Come First Serve, for if a socket is marked busy when a user process makes a request, the user process is put to sleep and awaken when the socket buffer is available. If multiple processors attempt this, they are awakened in pseudo-random order. Note, this is an area where *real concurrency* causes different, but semantically compatible behavior on a multiprocessor. *sleep()/wakeup()* issues are further discussed below.

5.5 Connect Semaphore

Often, protocol connection establishment is an asynchronous activity in protocol management. This is because, for example, protocol handshakes must be completed before the connection is completed. This is also true for disconnection and often a protocol engine requires that state information remain available until the disconnection is complete. In a monoprocessor kernel this is accomplished via *sleep()/wakeup()* semantics. In DYNIX this is accomplished by using semaphore structures. A connection semaphore is included in the socket structure and when a connect is requested, the protocol engine is called with a PRU_CONNECT request and the process is queued on the connect semaphore.

Note it is an easy mistake to assume that the above semantics describe simple replacement of *sleep()* with *p_sema()* and *wakeup()* with *v_sema(),* but this is not the case. Care must be taken in porting *sleep()/wakeup()* semantics to a real multiprocessor. In a monoprocessor, the awakened process does not run until the kernel relinquishes the processor. In a multiprocessor system, a *v_sema()* can make a process run immediately. In fact, the awakened process can start even before the return from the *v_sema()* call completes.

6. Things That Can Go Wrong

The purpose of this section is to give the reader an appreciation of the things that can go wrong when developing IPC in a multiprocessor system. The tasks is complicated by the objective to take advantage of as much existing code as possible, and absolutely maintain interfaces and user services. This is the objective of the DYNIX port, and it fundamentally is true to the UNIX 4.2bsd IPC implementation.

DYNIX adheres to the concurrency model described above as much as possible however, the model does not fall out of the port easily and great care is required to avoid multiprocessor pitfalls. Even so, it isn't proven until it actually works in a true multiprocessor system. Additionally, it must also be proven in a true networking environment since network processing happens as a result of other machines on a network. For this reason considerable stress testing is required to ensure completeness. The stress tests used must exercise heavily all of the error paths in a concurrent system. For the most part, this was done and needless to say, some errors were uncovered. The following discusses the types of errors that were encountered, how they were analyzed, and finally how they were fixed.

6.1 Mutual Exclusion

Mutual exclusion is never having to say %@#$^!!&^. It is the fundamental requirement to consider in multiprocessor systems. In fact, it is required in any system that allows multiple execution streams to modify the same data structures. In such systems, portions of the execution streams are sensitive to concurrent modification of shared resources. These form **critical sections** of execution during which concurrent modification must be avoided by mutually excluding other processes from conflicting critical sections of execution. Critical sections typically occur whenever shared data is modified. Some critical sections, particularly in a porting effort, are difficult to notice. For example, it is common practice to initialize local variables with their declaration (e.g. int foo = bar;). If the local variable is initialized using a shared data structure, the initialization must be moved to occur *after* the lock is acquired.

When mutex is required, it is critical for predictable system operation. Inadequate mutex can cause **race conditions** in which different execution streams "race" for a resource and if one looses, data is compromised. Because of race conditions, some errors are difficult to find because a system becomes nondeterministic. An error may not occur predictably after any particular sequence of execution although typically some "impossible" condition eventually occurs and the system fails. Most problems encountered in development of a multiprocessor system are due somehow to improper mutual exclusion.

6.2 Why the Monoprocessor Mutex Model Doesn't Work

Mutex is required even in monoprocessor environments if multiple streams of execution are supported. This is a common case, for example, in systems that allow interrupts and device handlers that modify shared memory areas. The UNIX monoprocessor model of mutual exclusion uses processor priority level (SPL) to mutually exclude one stream of execution from another. Streams of execution are blocked from critical regions by "raising" the SPL of the hardware. This prevents a process that executes at a lower or equal SPL from modifying a data structure until the SPL is lowered. For example, in UNIX4.2bsd code that requires mutex of network processing raises the SPL to SPLNET to prevent network code from asynchronously accessing shared structures as a result of unsolicited network input (or timer event). This is an example of a simple case of mutual exclusion requirements since one and only one processor still does just one and only one thing at a time. Obviously, actual concurrency is not generally achieved with a monoprocessor model.

A common pitfall in assessing the methods to use in porting a UNIX kernel to a multiprocessor environment is to assume that the use of SPL identifies all of the critical sections and simple replacement of SPL management with appropriate mutex calls is sufficient. It is not. DYNIX provides SPL-like capabilities on a per processor basis via the mutex calls. However, the SPL mechanism does not provide mutual exclusion of other processors executing at any other SPL. The SPL mechanism does provide simple mutex in a monoprocessor system, but it is by no means equivalent to a lock or gate type mechanism. The semantics of SPL management are quite different than the semantics of locks and gates. Simply replacing SPL calls with what seems to be appropriate locking and unlocking calls quickly causes dangling references, deadlocks, or other multiprocessor problems as discussed below.

6.3 Dangling References

A common error is that of **dangling reference.** This refers to the situation where a structure is referenced after the resources it uses are released. When multiple processes share a structure, the structure cannot be released until all processes are finished with it. Consider that after the resources are returned to the system, they can be used for something entirely different by another processor. The fundamental shared data structures in the IPC kernel are *mbuf's.* They are used for every dynamic storage need in the IPC kernel and managed separately from other kernel memory space. A released *mbuf* is placed on the front of the available *mbuf* free list and so typically is reallocated to another processor immediately in a multiprocessor system.

One of the dangers encountered in the port of a monoprocessor UNIX4.2bsd is that there exist dangling references that are harmless in a monoprocessor system. Various structures are released and later referenced within the same system call or network input process. Kernel structures are sometimes released and later used to reference, for example, a sleeping process that is waiting for a state change on the resource. The correctness of such coding practice is arguable even in a monoprocessor system, however it does not necessarily cause system failure. This is because the system call is typically executing at a raised SPL. The released *mbuf* is not reallocated in a monoprocessor until the system call lowers the SPL. Therefore, the referenced information is still useful even after the resource is released. This is not the case of course in a multiprocessor system since resources become available to other processors immediately upon being released.

Dangling reference can also be introduced with general mutex modifications. A simple example illustrates such a dangling reference. Consider the following generalized program structure:

```
LOCK resource;

switch(cmd) {
CASE 1:
        { }
CASE 2:
        { }
CASE N:
        { }
}

UNLOCK resource;
```

This appears straight forward enough and reasonably structured. However, if there exists one or more paths through the switch cases that releases the resource upon which the LOCK operation acts, the UNLOCK operation uses a dangling reference to a nonexistent resource. Since that resource is immediately available to other processes executing on other processors, the results are unpredictable. The UNLOCK operation above makes its state change to an *mbuf* that is being used by another processor for something else. Algorithms must accommodate cases where the referenced resource *disappears* during the critical section and therefore the lock which seems necessary to unlock must in fact not be unlocked due to dangling reference.

A side note in these cases is that the SPL is separately managed here. Mismanagement of an SPL can result in a processor becoming stuck at a particular, high SPL. In a multiprocessor system this might go unnoticed since there are other processors to take over the lower SPL tasks, but in a monoprocessor system being stuck at a high SPL can cause a form of deadlock discussed below. In DYNIX, SPL is usually managed via the *p_lock(), v_lock()* semantics, but in these cases where *v_lock()* cannot be called, *splx()* is used explicitly to restore SPL.

6.4 Deadlocks

Deadlocks refer to the phenomena where one or more processors are put into a position such that they can not make progress due to faulty mutual exclusion. Several types of deadlocks exist including sequential deadlocks, asynchronous deadlocks, and deadlocks involving multiple resources.

Simple **sequential deadlocks** often show themselves even in monoprocessor situations. They occur when a lock is acquired and later in the same execution path, without releasing the lock, another attempt is made to acquire the lock. In porting code this can happen due to use of common modular routines and macros. For example a resource close routine can be called from several places in the code some of which already possess appropriate locks and some that do not. A solution to this problem is duplication of the routine with appropriate locking semantics. This is done in a few places in DYNIX, but generally avoided if possible.

An **asynchronous deadlock** occurs some time after the fact, on a subsequent reference to the shared resource. This happens for instance if an error path out of a system call or network process does not properly release a lock or gate. In such a case, the lock is held never to be released and the next attempt to acquire the lock deadlocks. Note, that it is not usually the same sequence of code that left the lock held that ends up spinning on it. This makes debugging difficult without appropriate tools. One useful tool for this case is a circular buffer to record lock access activity. Such circular buffers were used in the development of DYNIX.

Deadlocks are more subtle with semaphores since an affected process essentially becomes inactive, waiting for a semaphore which is never incremented. Depending on the number of processes that share the semaphore, eventually all processes that access that semaphore end up waiting.

Another useful tool for deadlock detection is **processor lights**. The Balance 8000 displays LEDs on the processor boards. These lights are operated by the kernel scheduler. When the processor becomes idle, its light is turned off. When it becomes active, it is turned on. Experienced developers can determine a lot about the operation of a multiprocessor system by watching the lights. If a processor starts to spin on a lock, its light stays on brightly. If multiple processors deadlock, multiple lights stay on brightly. In systems of numerous processors, deadlock may not be obvious from the console or terminal keyboard, since it may only deadlock two of the available processors. In such cases, console and terminal response are unaffected and processors that are not deadlocked proceed. CPU intensive processes also keep the lights bright but typically occasionally flicker and often migrate from processor to processor. If all processes in, for example, a test suite become inactive due to improper semaphore management, all of the lights stay out. The method of using lights is often subjective, but proved useful in characterizing multiprocessor system behavior.

Deadlocks that involve multiple resources are more difficult to detect, and sometimes do not occur in a monoprocessor system since the critical sections are not really executed concurrently. The classic case of **multiple resource deadlock** involves two processes that require two shared resources, say locks. One process has resource A and needs resource B. Another process has resource B and requires resource A. This situation easily deadlocks unless care is taken to avoid it. One approach to avoid deadlocks like these is to always acquire resources in the same order. For example, if the above locks are always acquired A then B, then deadlock does not occur. Whichever process gets to A first, blocks any other process that requires both resources from getting resource B. In some cases it is possible to open a **window** during which another process can acquire both resources in the prescribed order. Allowing such windows is sometimes convenient but must be done with care since it introduces race conditions. Typically, a state must be reliably checked after the resource is acquired to ensure that its state did not change during the window. The process that allows the window indicates that it still has a reference to the structure so the winner of the race condition does not pull the resource out from under it. Consider, the state of the resource cannot be checked if the resource is deallocated. Windows also have performance considerations since they usually introduce additional mutex requirements.

There are also subtle deadlocks that do not occur in multiprocessor systems, but can occur in monoprocessor systems. For example, consider a process at a low priority level acquiring a resource that is eventually demanded by a process at a higher priority level. In the multiprocessor case, generally the lower priority process is eventually allocated processor bandwidth and allowed to complete execution of the critical section. However, the same situation executed in a monoprocessor system deadlocks due to the fact that the high priority process does not relinquish the processor to the lower priority process to complete the critical section. For this reason, testing of multiprocessor implementations must necessarily include testing for the case of N = 1.

6.5 Nested SPL's

In a monoprocessor UNIX system, SPL's can be "nested". This means that an execution path modifies an SPL, then later modifies it further. This is done in the kernel for instance due to use of modular routines and general purpose macros. Consequently it is possible to set an SPL to a particular level, and later execute a common subroutine which executes a similar SPL request. The affect in a monoprocessor system of the second SPL call is essentially NULL.

If the same resource is being accessed and the *spln()* calls are simply replaced with *p_lock()* calls, immediate deadlock occurs. It is possible to nest locks by referencing multiple locks but this encounters the multiple resource mutex management issues described above. Also note that in a monoprocessor system where nested SPL's are used for mutex, it is not always necessary to execute an *splx()* for every nesting level. Often it is sufficient to execute one *splx()* in order to return

to the user priority level. This is obviously not the case for nested locks since such an algorithm leaves locks held. This results in an asynchronous deadlock the next time the lock is referenced.

Nested SPL's do not always raise the SPL. It is possible to nest *splx()* calls which restore the SPL. In a monoprocessor system multiple calls to *splx()* to restore the same SPL are harmless, but if mapped one to one to UNLOCK operations in a multiprocessor system mutual exclusion is easily compromised. Observe that what occurs in the above situation is an extraneous unlock of a resource. This can be difficult to find if nested deep in the code because the damage does not occur while executing the erroneous execution stream. The erroneous execution stream essentially unlocks another processors lock and thereby opens up another processor's critical section.

6.6 IPC Calls to IPC Services

The IPC kernel also calls IPC services which introduces another source of potential deadlock. For example, a powerful notion in a networking system is the ability to access local services using the same addressing mechanisms as remote services. This is accomplished via a loopback path. Loopback allows local IPC identical to remote IPC. This means that the output side of the communications subsystem makes use of the services of the input side. This is done using the loopback driver which performs both as a sender and receiver. Loopback operates by taking packets from the transmit side of a network interface and placing them into the receive side of another network interface. From the protocol engine viewpoint, there is no difference to the data flow.

The 4.2bsd implementation also implements an Address Recognition Protocol (ARP) within the Ethernet driver which presents an interesting case. The network interface receiver interrupt accepts a data packet, determines that it is an ARP request, and calls an appropriate routine. The ARP data base is searched and it is determined whether or not a reply is required. If a reply is required a response is queued to the appropriate network interface transmitter which executes concurrently. The Ethernet output routine also queries the ARP data base to determine if address resolution is required. In effect, the Ethernet receiver initiates an Ethernet transmission. In the monoprocessor model, this by necessity is accomplished by a single processor at the same SPL and thus mutual exclusion is not an issue. In DYNIX, the transmitter and receiver execute concurrently and share the ARP data base.

6.7 Deadlock Avoidance Example - Pipes and Socket Peers

An example of how deadlocks are avoided in DYNIX is the implementation of pipes. Pipes are a particularly powerful, very popular UNIX capability. In the UNIX 4.2bsd kernel they are essentially a limited special case of the standard IPC mechanisms. Limited because they are unidirectional. In a monoprocessor implementation of UNIX 4.2bsd, multiple processes are involved with pipes, but not multiple processors. Therefore, the kernel can complete the data transfer from one user process, into kernel buffers *(mbufs),* and queue them for the receiving user process before the receiving user process starts receiving the data into its user buffers. However, in DYNIX both processes execute concurrently.

The *pipe()* system call creates two socket type file descriptors. One socket is maintained to keep track of the send side of the pipe and another is maintained to keep track of the receive side. Consider the straight forward approach. Process A is a producer of data to put into the pipe and process B is the consumer. Process A executes a *write()*. The AF_UNIX domain protocol engine locks the sender socket and finds the receive side. It then locks the receive side and places the data into it, unlocks it and notifies any processes waiting that data is available. A consumer process does a *read()*. A concurrently executing AF_UNIX domain protocol engine locks the receive side of the pipe in order to check the state and take data out of the queue. It then must update the send socket to notify it that the data is accepted in order to affect flow control. Notice that both processes acquire both resource locks in a different order. This results in deadlock the first time a process attempts to send data while another processor is receiving data from the same pipe. It is not necessarily appropriate to open windows in this case since the order in which the states are modified is important and potentially multiple windows are required on

every send or receive request.

DYNIX pipes avoid deadlock by sharing a mutex structure in something referred to as a **socket peer**. A socket peer is established at pipe creation time and contains the lock structure used to mutex both sockets. If simultaneous read and write requests are made, one blocks out the other while it quickly modifies the socket state. Note, data is copied between kernel space and user space without a lock being held. This is particularly significant when one considers that the user process' space can be swapped out in a virtual memory system such as that supported by DYNIX. It also means that typically the socket is locked only while the send and receive queue parameters are updated, not while data is being moved.

6.8 Deadlock Avoidance - Network Processing and Socket Peers

Socket structures are also accessed from the network concurrently. A protocol engine can be emptying the send queue while a user process is attempting to place a request into it. Protocol engines use the send queue to define flow control, and keep data in the queue until its delivery is assured. The queues can be modified by timer events or as a result of unsolicited network control messages. The protocol engine also references a protocol control block that contains addressing and state information. The protocol control block is used by the protocol engine to find the associated socket structure. It is cumbersome to simply provide a lock on the protocol control block and a lock on the socket because access to these structures is both network to kernel and kernel to network. This means that sockets are accessed both through protocol control block reference and through socket references. Notice that this presents multiple resource deadlock concerns similar to those discussed above.

The approach taken in DYNIX is again to define a **socket peer** mutex structure similar to those used for pipes only now the sharing entities are socket management and network protocol engines. The socket peer is created at protocol "attach" time and is used to mutex both the socket and the protocol control block. Therefore when critical sections are necessary in the protocol engine related to a particular socket structure, user processes that access that socket are locked out. Typically this is for a short time until data gets queued or dequeued from the interface. Note, that unsolicited data also implies unsolicited control data, and the shared structures can be modified, even deleted due to a network input. Therefore, once a protocol control block is found by the protocol engine, its socket peer mutually excludes the user from changing its state until its action is completed. Other user process socket structures can access other sockets and other network functions while a protocol engine works on a particular socket. Also other protocol engines can be executing.

To manage creation and deletion of socket peers, a reference count is maintained per socket peer. The socket peer can be referenced by the user process space, by the network binding space, by the network input space and by special events such as timeouts and unnatural termination requests. An example of an unnatural termination requests is the closure of a route to a foreign hosts. Recall that in the Internet this results in peer level IP layers exchanging Internet Control Messages (ICMP) which results in all currently active connections to a suddenly unreachable destination to be aborted.

To appreciate the issues in concurrent processing consider the worst case in this model. The user process, in fact several user processes, can be attempting to close the resource, due to termination for example. The route could have become unreachable such that ICMP messages have directed its abortion. A timer event could expire indicating a complete disconnection which results in the deletion of the connection. All of these events can occur simultaneously and be assigned to different processors to execute concurrently. Resources must remain available until the last reference is made to them to avoid dangling reference. Obviously, a monoprocessor implementation does not have to consider all of these events occurring simultaneously since it can essentially serialize the requests and deny or ignore those that it schedules after deletion.

The port to DYNIX of the IPC is very careful to arrange deletion algorithms such that references to a resource are not made after the resource is removed. This means for example that when a socket is deleted, the reference count of the socket peer is checked to ensure that there are no longer any references to it before it is released. If there are references, then it is unlocked in order to allow the last references to be removed. This occurs for example if the socket is being closed while the network is processing an input packet (control or data) that is bound for it. The algorithms discover later that the socket has been closed since the protocol engine that executes on behalf of the user to close the socket, typically changes the state of the protocol control block to indicate disconnection. The process that makes the last reference to the socket peer releases the resources. Note the mutex considerations here. As described above, it is not proper to UNLOCK a lock structure that has been deallocated. The general socket-peer deletion algorithm becomes:

```
LOCK socket peer;
release appropriate resources;
increment socket peer reference count;
do appropriate protocol action; /* which may cause deletion */
decrement socket peer reference count;
if(last reference)
        release socket peer resources;
        restore appropriate SPL;
else
        UNLOCK socket peer resources;
```

Also as in the case of the generalized system call example where the lock structure is released, returning to the appropriate SPL is considered. Since the DYNIX *v_lock()* interface returns to a specified SPL, those semantics are emulated in cases where *v_lock()* is not used.

6.9 Concurrency and Shared Lists

A common method of sharing information is in the form of lists. These can be managed as queues, or static or dynamic data structures. Examples of lists maintained in the kernel include lists of data streams, pending connection requests, and protocol control blocks used for polling, address binding and synchronization of system resources.

A example of this is the list of protocol control blocks maintained for active TCP ports. These ports can be in various states such as newly created, not yet connected, listening for connection requests, connection established, partially disconnected and finally disconnected and on its way out. The list is headed by a TCP Control Block structure and consists of a linked list of TCP Control Blocks that contain addressing and state information. When a TCP packet arrives, it is first demultiplexed by the network interface (viz. Ethernet) driver and placed into the Internet input queue. The Internet input queue handler, *ipintr(),* executes concurrently at a lower SPL. The Internet protocol engine quickly recognizes a TCP packet and calls the *tcp_input()* routine. The *tcp_input()* routine scans the TCP control block list to determine if it knows about the destination specified in the packet (i.e. the port number). The protocol control block list is concurrently accessed by user processes that add and remove references and also by the timer event routines which periodically scan the list to change the protocol control block state (e.g. decrement a time counter, execute a protocol time out event,..). Also consider the *sonewconn()* case which results in the input side of a protocol engine adding a new protocol control block to the list. Obviously, both the protocol control block list and the particular protocol control block require mutex.

A socket peer approach is not appropriate in this case since there are many peers in the list. Therefore the straightforward approach is for *tcp_input()* to lock the TCP control block list, find the appropriate protocol control block and lock it. However, note what happens if the

protocol control block is associated with a socket that is in the process of being deleted by a user process request. The kernel on behalf of the user process locks the socket and the protocol control block (using their socket peer), and then locks the list in order to detach the protocol control block from the list. This creates another deadlock.

DYNIX avoids deadlock here by assuring that in those cases that require both the list and elements in the list to be locked, the locks are always acquired in the same order. First the list is locked and then the element. This order is chosen because locking the element first and then the list results in many more windows being opened by processes that traverse the list. Opening a window allows a race towards acquiring the lock for the element. Care is required that this race can be lost and state information not be compromised. When an element is being deleted, its state is completely updated before a window is opened in order for it to be detached from the list. This avoids access by other user process requests. To avoid access from the list processing functions, further indication is used. Each entry contains a pointer to the head of the list. A NULL value in this pointer indicates that the entry is on its way out. If a processor is traversing the list and wins a race for the lock of a control block that is on its way out, the NULL head pointer value indicates that there is no processing appropriate for this entry. Care is taken not to reference protocol control block state information after it is possible to delete it and information such as the pointer to the next element in the list is maintained across the call. The pointer to the next element is acquired with the lock held to avoid a dangling reference when the control block is deleted. The control block is then unlocked, thereby allowing the deletion to continue.

Some actions of list processing result in deletion of the element under consideration. This means for example that list processing calls list deletion routines. Alternate detachment routines are implemented which detach elements with the list locked. If convenient, these routines are called, otherwise, windows are permitted and algorithms ensure that no dangling reference occurs. The list processing increments a reference count on the lock structure (e.g. the socket peer) in order to avoid it being deleted along with the control block.

6.10 Concurrency and Time Outs

Time waits for no LAN! Many functions of practical protocol engines depend upon the system (i.e. the kernel) to keep track of time and its relation to state information. It accomplishes this by issuing a time out event at regular intervals. Specifically, transmissions time out and are assumed lost, retransmissions are submitted, round trip delays are calculated and connections are abandoned due to certain time limits being exceeded. In a monoprocessor environment everything literally stops during a timeout and the entire state of the protocol engine examined and acted upon. This is not the case in a real multiprocessor environment and considerable attention is required for protocol timeouts.

At each time event, the domain structure is referenced to find the appropriate *protosw*[] array and the time event routine for each protocol. There are two such events, the fasttimo event every 200 ms. and the slowtimo event every 500ms. Such events are used to calculate protocol timeouts, etc. Note that not all protocols maintain states based on time events. For example, non-guaranteed data gram delivery offered by the Internet UDP protocol specifies NULL time out event routines in the *protosw*[] entry.

The TCP time out routines are also concerned with list processing similar to the *tcp_input()* routine. Time out events can result in deletion of protocol control blocks and so the reference counting algorithm is used. The TCP timeout routines additionally avoid spinning on protocol control block locks. It does this by using the conditional *cp_lock()* interface. The assumption is that if a protocol control block is locked, it is undergoing state change that most likely includes timer state changes. This is true in most instances, and it is noncritical if a timer event is missed.

6.11 Cached Temporary Data

The term **cached temporary data** refers to a situation where shared structures are used to temporarily store data during a system call that has network interrupts blocked out on a monoprocessor system. The shared structure is restored to the appropriate value before network processing is continued. A specific example of this is the use of globally referenced structures used to hold temporary values. These structures are moved to the local parameter scope (i.e. the stack) to avoid sharing them when not necessary. A more subtle example of this is the use of UDP protocol control blocks to store the address of an outgoing datagram. This information is passed to the UDP transmit routines and used to create the UDP packet header. After the UDP datagram is created and queued for output, the protocol control block is restored. This does not work correctly in a multiprocessor system because concurrent references are made to the protocol control block by the network input routines searching for address matches. For the time that the protocol control block is being used for temporary storage of temporary binding information, addresses do not match. In the case of UDP data grams, this means that they are not delivered. This situation is easily tested by having multiple UDP clients sending datagrams to a single UDP echo server. This typically results in simultaneous send and receives and consequently datagrams are frequently missed.

6.12 Data That Corrupts

Improper mutual exclusion can cause situations where network interfaces put data where it shouldn't go. This is particularly true when asynchronous DMA is supported and exacerbated by shared memory in multiprocessor systems. Corrupting data buffers can result in application programs failing, but it can also result in protocol failure since at some level, protocol control information (e.g. headers) is generic data. Furthermore, since *mbuf*'s are also used for many IPC kernel structures they too can be corrupted by faulty data placement which can cause unpredictable behavior. There are safeguards against errant kernel code modifying code space, but these safeguards do not usually apply to system i/o such as DMA from network interfaces.

In a multiprocessor system that supports DMA, it is sometimes necessary to narrow down such a problem to the network subsystem. It is helpful that the kernel typically uses only IPC kernel buffers, *mbufs,* for IPC kernel managed data. It is also helpful that all network buffers start on an *mbuf* byte boundary (modulus 128) and all DMA requests are *mbuf* size (128 bytes) or less. These observations can be applied to verifying that a network interface is compromising system memory.

Another approach to debugging errant data transfer by a network interface is to use tests that transfer identifiable patterns. This approach helps in general to debug data movement implementations but there are special considerations particularly applicable to network system development. Patterns should be chosen with care. They should be easy to recognize in an environment where lots of data is present. In network system development, it is often necessary to view *exactly* what goes on to the wire. Therefore patterns should be readily recognizable in a primitive form of representation (viz. hexidecimal). For example, it is useful if the patterns "spell" recognizable "words" using hexidecimal digits. Networking development is often concerned with bit and byte ordering since it deals with heterogeneous machine architectures [Kirrmann 1983] and network canonical forms. This implies for instance that test patterns should not be palindromes and be able to indicate byte swapping errors. There are several interesting hexidecimal patterns which meet these requirements.[9] A particular pattern chosen in DYNIX development is hexidecimal FEED FACE DEAD BEEF. Using this pattern in various portions of network data packets assisted the tracking of data flow from machine, to network, to machine. If byte swapping errors occur, FEED FACE becomes EDFE CEFA or CEFA EDFE (depending on whether the error is on a short or long). If network data is incorrectly placed in the destination and something breaks, FEED FACE DEAD BEEF is evidence enough to suspect problems in the flow of

9 Try egrep '^[a-f][a-f][a-f][a-f]$' /usr/dict/words.

network data.

7. Conclusions

DYNIX is not *the* solution to loosely coupled functions on a tightly coupled engine, but it certainly is *a* solution. It works and works well. The problems encountered in porting the described UNIX 4.2bsd IPC model to the described DYNIX multiprocessor model are similar to problems encountered in any multiprocessor system. However, as this paper describes, the nature of networking introduces considerations such as unsolicited data and remote initiation of shared resource modification that exacerbate common multiprocessor problems and manifests them in ways particularly applicable to networking. The lessons learned and the solutions implemented described in this paper are considered useful for further research and development in general multiprocessor technology.

References

[Bach 1984]
Bach, M. and Buroff, S., "Multiprocessor UNIX Operating Systems", AT&T Bell Laboratories Technical Journal, Vol. 63. No. 8, October 1984.

[Beck 1984]
Beck, R. and Kasten, R. "Multiprocessing with Unix", Systems and Software, pp. 135-138, October 1984.

[DEC 1982]
"The Ethernet A Local Area Network Data Link Layer and Physical Link Layer Specifications", Version 2.0, Digital Equipment Corporation, Maynard, MA, Intel Corporation, Santa Clara, CA, and Xerox Corporation, Stanford, CT, November 1982.

[Dijkstra 1965]
Dijkstra, E., "Solution of a Problem in Concurrent Programming Control", Communications of the ACM, Vol. 8 No. 9, pp. 569-78, September 1965.

[Fielland 1984]
Fielland, G. and Rodgers, D., "32-bit Computer System Shares Load Equally Among Up To 12 Processors", Electronic Design, September 6, 1984.

[Kirrmann 1983]
Kirrmann, H., "Data Format and Bus Compatibility in Multiprocessors", IEEE Micro, August 1983.

[Leffler 1983]
Leffler, S., Joy, W. and Fabry, R., "4.2BSD Networking Implementation Notes", Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, July 1983.

[Leffler 1983B]
Leffler, S., Joy, W. and Fabry, R., "A 4.2bsd Interprocess Communication Primer", Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, July 1983.

[O'Toole 1985]
O'Toole, J., Torek, C. and Weiser, M. "Implementing XNS Protocols for 4.2bsd", Proc. of USENIX Association, Winter Conference, pp. 90-97, January 1985.

[Requa 1985]
Requa, J. " Unix Kernel Networking Support and The Lincs Communications Architecture", Proc. of USENIX Association, Winter Conference, pp. 98-103, January 1985.

[Sequent 1984]
"Balance 8000 System Description", Sequent Computer Systems, Inc., 1984.

[Shoch 1979]
Shoch, J. and Hupp J., "Performance of an Ethernet Local Network: A Preliminary Report", Proc. of LACN Symp., May 1979.

[UNIX 1983]
"UNIX Programmer's Manual (4.2BSD)", Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, August 1983.

[Zimmerman 1980]
Zimmerman, H., "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection", IEEE Trans. on Comm., April 1980.

Lightweight Processes for UNIX[†] Implementation and Applications

Jonathan Kepecs
Sun Microsystems Inc.
2550 Garcia Ave.
Mountain View, CA. 94043
415-960-7279
sun!kepecs

1. Introduction

A *lightweight process* (lwp) is a process that maintains little associated state so that context switches and process creation are relatively inexpensive. Interprocess communication (IPC), i.e., the exchange of control and data between lwp's must also be efficient to make using them attractive. Typically, this implies that lwp's share memory to minimize data transmission costs and the amount of process state, although, since message—passing can be implemented with shared memory, messages are not excluded from the repertory of lwp IPC mechanisms.

One way to obtain inexpensive processes is to provide language—level support. Languages such as Modula-2 [1], Mesa [2], C [3] and Simula [4] have incorporated processes either as part of the language or via library facilities. Because processes are an elegant structuring tool, we would like to employ them to facilitate the design of applications programs as well as the design of the UNIX kernel. Charlotte [5] and SUBMARINE [6] are examples of operating systems that have used language—level processes to construct large operating systems. A major goal of our work is to provide a single, integrated set of lwp primitives for use in structuring both user and kernel programs. This paper describes ongoing research at Sun investigating the use of lwp's in the UNIX environment.

1.1. Applications

The present Sun UNIX kernel is essentially a monolithic monitor. When the kernel has a task to do, it must borrow a thread of control from a user process. For example, when a client wishes to perform non-blocking I/O, the kernel must remember state associated with the I/O and require the client to provide a thread for controlling the data transfer. The client typically does this by using a blocking *select()* system call and reissuing the I/O request when the select returns. Kernel lightweight processes could provide a convenient place to localize the I/O activity and state information without having to steal a control thread from the client. A client could poll for, or be notified about, I/O completion instead of having to retry the operation.

Another example of where lightweight processes in the UNIX kernel could be useful is in the Sun Network File Server (NFS[‡]) [7]. To implement read—ahead and write—behind, several daemon processes are required because a client is blocked on a physical data transfer while other remote requests are arriving. In order to obtain threads of control, user processes are created which use a special system call to enter and remain in the kernel. It would be much cleaner and possibly more efficient to have the threads of control available in the kernel without having to obtain them in such an indirect fashion.

Because a process is a self—contained unit, it is easily identified as a pageable entity. As the number of facilities available in the UNIX kernel increases, so does the size of the kernel and the desirability of paging infrequently used portions of the kernel. Further, because a process

[†]UNIX is a trademark of Bell Laboratories

[‡]NFS is a trademark of Sun Microsystems

encapsulates the details of a portion of kernel code cleanly (a process is accessible to other processes only through a narrow IPC interface), designing the kernel as a set of cooperating processes can improve understandability and robustness. If processes can be made truly lightweight, the cost of using them will not be a deterrent to their use in reorganizing the kernel.

At the user level, the availability of lightweight processes has important ramifications. Many modern programming languages have a notion of multitasking built in, so a general—purpose facility for supporting more than one thread of control within a single address space would be a useful addition to UNIX. The implementation of major subsystems such as the Sun window system and the Sun remote procedure call (RPC) mechanism could be improved substantially with an effective lwp mechanism.

One possible window system design calls for a collection of lwp's, with one lwp controlling each window. A client process reading from or writing to a window would make requests only of its controlling lwp, instead of a large monolithic controlling process which must maintain state about each individual window explicitly.

A problem with the present Sun RPC mechanism is that although it supports the notion of a server handling multiple concurrent requests, it must be programmed with a single thread of control. Thus, a server blocks while *selecting* an incoming request. When a request arrives, the appropriate procedure is called. No new requests can be processed until the procedure completes, and this may be unfortunate if the procedure itself blocks, possibly *selecting* I/O itself. The natural way to implement the service mechanism is to create a separate thread of control when a procedure is called. This method is not only cleaner, it also permits greater concurrency.

2. Kernel Design

We propose to provide two separate, but interacting versions of the lwp mechanism: user and kernel. The kernel mechanism is meant to provide structuring for traditional kernel operations such as accessing physical devices, and the mechanism for supporting traditional "heavyweight" processes (*hwp's*). The user mechanism is intended to support multiple threads within a single hwp. Currently only the user mechanism is implemented. However, we intend that the same primitives will be available in both versions. This has the advantage that it becomes easy to write kernel utilities as user processes and then move them directly into the kernel. As a result it is possible to debug kernel code without worrying about crashing a running system.

Our hypothetical kernel consists of a resident *nugget* that provides kernel-level lwp's (*klwp*'s), IPC between pairs of klwp's and between a klwp and a hwp, physical memory allocation and interrupt handling. Lwp's have scheduling priorities. Within a priority level, scheduling is non—preemptive. However, a lwp with a higher priority will preempt one with a lower priority. Each lwp has a queue of messages associated with it. A lwp is eligible to execute if its message queue is nonempty.

One of the lwp primitives causes processor—specific events (e.g., traps and interrupts) to be mapped (by a procedure registered with the nugget) into IPC calls to specific klwp's. The message associated with interrupt—triggered IPC will contain volatile device information, such as the current input character from a tty, which the mapping procedure has enqueued. An efficient synchronization algorithm is possible if the mapping procedure can not itself be interrupted by the same device: Only the mapping procedure may enqueue and dequeue device information. When a driver klwp is informed (by an IPC issued from the mapping procedure) of the presence of a new queue entry, it will read the information and mark it as having been read. The mapping procedure is responsible for dequeueing marked items upon subsequent interrupts. Thus, the interrupt handling procedure functions as a monitor except that locking is done only when it is required (at interrupt time) instead of as a preventative measure as is the usual case with monitors.

IPC between requesting and serving klwp's consists of the requester sending a message to the server, thereby making the server eligible to execute. The requester provides the server with input parameters via a page—aligned input buffer which is marked read—only and

double—mapped into the server address space. Results are returned by a similar buffer which is made available to the server klwp for writing. If the hardware permits, the result buffer should be marked write—only to the server. By double—mapping buffers, no copy costs are incurred. The nugget provides a memory allocation service which provides page—aligned memory chunks.

One of the klwp's is the user scheduler. It manipulates user space (klwp's reside in kernel space) and implements hwp's by periodically altering the contents of user space. The scheduler also provides some support for the user lwp (*ulwp*) mechanism (although much of the ulwp support can be done at the language level).

The nugget executes klwp's as long as a ready one (i.e., a lwp with nonempty message queue) exists. When there are no ready klwp's, the nugget executes whatever program currently resides in user space (this is the idle loop in the system). When the scheduler klwp is made eligible to run by some event (a clock tick triggering an IPC call to the scheduler, for example), it performs a user—level context switch simply by changing the program that currently occupies user space. When the idle loop resumes, a new user (heavyweight) process will be executed. The user process can make requests for service by issuing IPC calls to server klwp's.

3. LWP Primitives

In order to explore the utility of lwp's in UNIX, we began with the more tractable task of implementing them within a hwp by providing a runtime support library for C programs. An important goal was that lwp's be able to use the Sun RPC mechanism [8] to communicate with other UNIX processes. Naturally performance is of major concern and we discuss the performance of our implementation in § 3.3.

To use the lightweight process mechanism, a C program must be linked with two libraries of routines: one which provides the lwp primitives (*nugget library*), and another which provides a non—blocking version of the standard UNIX primitives such as read and write (*syscall library*). The latter is needed because some UNIX primitives (e.g., reading from a tty) can block arbitrarily long, preventing other ulwp's from executing. The syscall library however does not require any additional mechanisms beyond those provided with the nugget library. A client program supplies a procedure with a well—known name. This procedure becomes the first thread to execute when the program runs. We will refer to the collection of lwp's executing within a UNIX (heavyweight) process as a *pod*.

There are eleven primitives currently available to the client. To simplify the discussion, error return values from the primitives are omitted. When a message is received, a client will be presented information about the message, including the identity of the sender, the size of the buffers, and a unique identifier that refers to the message. We refer to this information as the *tag* of the message:

```
typedef struct msg_t {
        int msg_type;                   /* type of invocation */
        int msg_tid;                    /* unique tid */
        int msg_pid;                    /* caller's pid */
        int msg_arg;                    /* extra argument */
        unsigned char *msg_obuf;        /* caller's args to callee */
        int msg_ocnt;                   /* byte count of obuf */
        unsigned char *msg_ibuf;        /* callee's args to caller */
        int msg_icnt;                   /* byte count of ibuf */
}msg_t;
/* msg_type values */
#define T_RECEIVE 1   /* indicates a message has arrived */
#define T_REPLY 2     /* indicates a reply has been received */
#define T_DEAD 3      /* indicates that target of a send is dead */
#define T_SIGNAL 4    /* indicates that a signal has been received */
#define T_CREATE 5    /* indicates first invocation of a process */
```

```
int
create(pname, task, prio, stacksize)
        char *pname;
        int task();
        int prio;
        int stacksize;
```

creates a new lwp with a stack of size "stacksize," bound to the procedure "task," the scheduling priority "prio," and the identifier "pname." The process id (*pid*) of the new lwp is returned. This pid is distinct from the pid returned by the UNIX *fork* primitive. A newly created process will receive a message of type T__CREATE.

```
kill(pid)
        int pid;
```

destroys a lwp. Any lwp expecting replies from a destroyed lwp will be notified. A lwp is killed automatically when it exits.

```
int
discover(pname)
        char *pname;
```

locates the pid bound to the name "pname." If pname is NULL, the pid of the caller is returned.

```
int
send(pid, arg, prio, out__buf, osize, in__buf, isize)
        int pid;
        int arg;
        unsigned int prio;
        unsigned char *out__buf, *in__buf;
        int osize, isize;
```

causes a message to be enqueued for process "pid" with receiving priority "prio." "out__buf" is a buffer for transmitting parameters to the receiver, and "in__buf" is a buffer for receiving return results from the receiver. "osize" and "isize" are the sizes of these buffers. While it is intended that these buffers be page—aligned and protected (to the receiver, out__buf is made read—only and in__buf is made write—only) in the klwp implementation, they are completely unprotected in our current implementation. "arg" is an integer argument passed along to the receiver, and may be used to help screen incoming messages. *send* returns a "transaction id" (tid) to the caller. The tid uniquely identifies each *send*. *send* is non—blocking. A client can a simulate blocking *send* by issuing a *send* immediately followed by a *receive*.

```
msg__t *
receive()
```

suspends execution of a lwp until a message has arrived for it. *receive* returns a message descriptor that identifies the current message.

```
reply(tid, prio)
        int tid;
        unsigned int prio;
```

sends a response to the sender of message "tid" indicating that the message bound to "tid" has been read. The priority of the reply message is given by "prio." A reply appears as a message to the sender with "msg__type" set to be T__REPLY instead of T__SEND. If a process dies before issuing a *reply* to a message, the sender receives a message from the system of type T__DEAD.

```
yield(pid)
        int pid;
```

resumes process "pid" at the point where it last issued a *receive* or *yield*. If pid is 0, control is yielded to any eligible process (i.e., one with a message awaiting delivery with a process priority greater than or equal to that of the yielding process). If there are no eligible processes, *yield*(0) is a no—op. If "pid" is non—zero, *yield* functions as a coroutine *resume*.

```
handle(event, slavepid, prio, persistence, direction, fd)
        int type;
        int event;
        int slavepid;
        unsigned int prio;
        char persistence;
        int direction;          /* READING, WRITING, EXCEPTION bits */
        int fd;                 /* which descriptor is being selected */
```

is used to map UNIX events (signals) into *sends* to the lwp executing the *handle*. "event" is the UNIX signal being caught. "slavepid" is the pid of the lwp which is running at the time an event occurs. Typically, "slavepid" is the same as the pid of the process issuing the *handle*, but it need not be. This feature is included primarily for the klwp implementation because it is necessary to have a mechanism to handle faults on behalf of hwp's. For example, the scheduler can handle the memory management faults of the user hwp. As another example, a high—priority ulwp can be dedicated to handling asynchronous events such as a keyboard interrupt for other lwp's in a pod. "prio" is the priority of the message (of type T_SIGNAL) that will be delivered to the handling process. "persistence" is a flag that indicates whether the *handle* is to remain in effect permanently or once only. "fd" and "direction" are parameters to the special signal SIGSELECT. This signal is a special pseudo—signal provided by the ulwp implementation. A ulwp that *handle's* SIGSELECT on a descriptor "fd" will be invoked with a message indicating when that descriptor is ready for reading or writing or that an exception has occurred, depending upon the value of the "direction" bits. The "msg_arg" field of the corresponding message tag will indicate which fd has been selected. Any descriptor handled for SIGSELECT will automatically be marked non—blocking with the *fcntl* UNIX system call. In the klwp version, the "direction" and "fd" parameters are replaced with a procedure that tells the nugget how to map device information into a *send*.

```
unhandle(event, slavepid, fd)
        int event;
        int slavepid;
        int fd;
```

removes the effect of a *handle* operation. A given signal may be handled by more than one lwp, so *unhandle* has an effect only on the *handle* bound to "slavepid."

```
mask()
```

allows a lwp to mark and return the current message to its queue of pending messages. A *masked* message will not be delivered to a lwp until it is *unmasked*. If a client has no room for a message, *mask* is useful to achieve flow control.

```
unmask(type, arg, tid)
        int type;
        int arg;
        int tid;
```

allows a lwp to unmark a message or class of messages that have been previously masked. To screen messages, a lwp can *mask* any message it is not interested in, and *unmask* all messages after the message it wants arrives. "type" is the type of the message (T_REPLY, T_SEND, etc.) being unmasked. "arg" is the argument field associated with the message and "tid" is the

transaction id of the message. If every parameter to *unmask* is NULL, *all* messages previously masked by the unmasking lwp are unmasked. By allowing the client complete control over message screening, we avoid difficulties due to the necessity of evaluating conditions in a lwp's referencing environment. We also simplify the nugget library implementation. A disadvantage is that the client must be bothered to inspect messages that it may not be interested in at the moment.

3.1. An Example

We present here a simple example of a client program that uses the lwp primitives. This program transfers a buffer from one lwp to another using the IPC mechanism provided. Since ulwp's share memory, such a transfer could be achieved more efficiently by other means, so this example is meant to be illustrative only.

```
/* these values are in shared memory */
int child_pid = 0;
char outbuf[100];
char inbuf[100];

task()
{
        /* these values are private to each lwp */
        msg_t *curmsg;
        register int i;

        (void) receive();      /* discard creation message */

        /* make child process */
        if (child_pid == 0) { /* parent */
                child_pid = CREATE("child", task, 1, 1000);
                for (i = 0; i < sizeof(outbuf); i++) {
                        (void) send(child_pid, 0, 0, &outbuf[i], 1, 0, 0);
                        (void) receive();
                }
        } else {        /* child */
                for (i = 0; i < sizeof(outbuf); i++) {
                        curmsg = receive();
                        inbuf[i] = *curmsg->tag_obuf;
                        reply(curmsg->msg_pid, 1, curmsg->msg_tid);
                }
        }
}
```

When the nugget runtime library is initialized, "task" is created as the first lwp. The task begins by creating a child process that uses the same code. The parent then sends a message to the child containing the next character from "outbuf" and waits for a reply from the child. The child waits for a message from the parent, adds the character it receives to its own buffer, and replies to the parent. Notice that although no parent—child hierarchy is built into the lwp primitives, such a hierarchy can be constructed artificially. Also, we could have used two separate procedures instead of sharing the same code.

3.2. Implementation Notes

The syscall library provides access to UNIX system calls and Sun RPC commands. Some UNIX commands, such as read(), write(), sendto(), recvfrom(), and select(), can block for an arbitrary period of time. Since scheduling of ulwp's is not done on clock interrupts, such a command could prevent ulwp's from executing. To avoid this problem, these UNIX commands are handled specially. Each blocking UNIX command is mapped by the library into a *handle* on the descriptor(s) being accessed using the special SIGSELECT pseudo—signal, followed by a *receive*. If the *receive* returns and the message obtained is not the SELECTED one, it is masked and the *receive* is tried again. When the *receive* returns successfully, all pending messages are *unmasked* and the library version of the UNIX command returns. Thus, to each ulwp, a UNIX command appears to be blocking while not affecting the execution of other ulwp's in its pod. Timers (i.e., setitimer()) are also implemented by the user library because UNIX does not support more than a single timer per process.

Ulwp's communicate to other UNIX (heavyweight) processes via Sun RPC (see [8] for details). Since some RPC primitives are blocking and the RPC library itself uses UNIX system calls, it was necessary to modify the RPC library to use the syscall library and to make a few minor changes. The routines to register and unregister a service now include a hook to bind the sockets used by a service to the particular ulwp providing the service. Thus, when a server ulwp blocks to await requests, it is waiting for activity only on the sockets that it registered, not the sockets used by the entire pod.

For the most part, the ulwp runtime system avoids using UNIX system calls because such calls are relatively expensive. However, there are some features that we would like to make available as an option but are unable to because of limitations in UNIX. There is no protection of messages or values on a ulwp stack from corruption by other lwp's. Also, lwp stacks are not permitted to expand beyond the fixed initial allocation specified by *create*. A flexible interface to the UNIX virtual memory management system would be required to remedy these problems. Also, the operation to make descriptors non—blocking does not always work (for example, writing to a tty) so it is not possible to make the syscall library effective in every case.

Because all objects used by a client are marked non—blocking, there is a danger that the client will terminate abnormally, leaving some objects non—blocking with possibly disastrous effect. For example, if stdin is so marked, when the shell tries to read from stdin, it gets an error (EWOULDBLOCK) and dies. There is no really good way to fix this problem unless UNIX is modified to allow one to mark *descriptors* rather than the underlying *objects* as non—blocking.

3.3. Performance Results

Because performance is a critical issue in determining the usefulness of lightweight processes, we took some care to achieve a good result. For example, locking message queues against signals can be expensive because a UNIX system call is required. Therefore, we keep two separate queues: one for normal messages and another for messages which are *handled* and therefore triggered by signals. When the scheduler checks to see if a given lwp is eligible to run, it looks at both queues. If the signal queue is empty, there is no need to lock the queue, dequeue an entry and unlock the queue. In order to keep the implementation as portable as possible however, we implemented almost all of the runtime in C instead of assembly language.

Because the support for lwp's is provided by a runtime library instead of being built into the language, some optimizations are not possible. For example, there is no convenient way to know (with our present C compiler) what registers are actually being used at the time of a *receive* so all registers are saved. Nonetheless, we feel that performance of our primitives is acceptable. For example, on a Sun—2, the UNIX fork() requires about 35.6 ms whereas our CREATE primitive requires about 0.2 ms. The nugget library is also small: it only adds about 4300 bytes to a user program. To get a feeling for the actual performance of our system, we devised the following benchmark (all numbers are for a diskless Sun-2 with 2 megabytes of memory): copy a buffer from one process to another synchronously in coroutine—fashion, so that a context switch

is needed to move each character in the buffer. Our results are summarized in the table below.

Data Transfer Method	Control Transfer Method	Time (ms)
Parameters	Procedure Call	0.04
Shared Memory	Yield(pid)	0.16
Shared Memory	Yield(0)	0.52
Local Messages	Send-Reply	0.72
Unix Filesystem	Unix pipes	4.7
Remote Messages	Sun RPC	20.0

We feel justified in supporting two separate but related forms of IPC: the send-reply paradigm for lwp's communicating between themselves and the RPC paradigm for communication between lwp's and hwp's. The former is very inexpensive; the latter provides the full power of RPC between hwp's including authentication and parameter marshalling. Naturally, regular procedure call should be used within a pod whenever the synchronization facilities of IPC are not needed.

4. Conclusions

We have described a lightweight process mechanism which can be used to provide support for UNIX kernel development and user applications programs. As part of our investigation, we have implemented lightweight processes within a normal UNIX process by providing a runtime library, and reported on the performance of our implementation.

Our current design assumes a single processor. Other researchers [9, 10] are investigating the use of multiprocessors with shared memory to allow individual lwp's within a pod to execute concurrently. Our proposal does not provide support for such hardware but we are presently studying how the availability of a multiprocessor would affect the lwp primitives.

We provide a message—based scheme for interprocess communication and synchronization. Synchronization can be achieved by interposing a scheduling process between cooperating processes. By using a double—mapping technique we will be able to obtain the efficiency of shared memory, while providing the disciplined access to procedures and data provided by servers and messages. As this scheme is ideal for crossing addressing domains, we plan to implement RPC directly with the send—receive—reply primitives.

There is considerable motivation to provide monitors and condition variables as primitives even though monitor—based and message—based systems are duals of each other [11]. First, because lwp's share memory, monitors may be more inexpensive than server processes for concurrency control within a pod. This is especially true in the case where the monitor is not busy when entered, as no context switch is needed. Second, many programmers are used to a monitor—based style of programming. The UNIX kernel, for example, essentially provides monitor—like primitives such as sleep() and wakeup() which are analogous to waiting for a condition and notifying a process of a condition, and spl() which provides the kind of locking that a monitor uses while evaluating condition variables. As a result, we are considering adding monitors and condition variables to the set of lwp primitives.

5. References

[1] N. Wirth. *Programming in Modula-2, 2nd edition,* Springer-Verlag (1982).

[2] J. G. Mitchell, W. Maybury, and R. Sweet, "Mesa Language Manual Version 5.0," CSL-79-3, Xerox Corporation (1979).

[3] A. G. Fraser, "C language routines for multi—thread computation," TM 79-1273-4, Bell Laboratories (1979).

[4] G. M. Birtwistle, O. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin,* Petrocelli-Charter (1973).

[5] R. Finkel and M. H. Solomon, "Part IV of the first report on the crystal project," TR-499, University of Wisconsin technical report (1983).

[6] J. Kepecs and R. Sandberg, "The SUBMARINE operating system," unpublished technical memorandum, University of Wisconsin (1979).

[7] R. Sandberg. *Design and implementation of the Sun network filesystem,* Usenix Conference (1985).

[8] Sun Microsystems. *Remote Procedure Call Reference Manual,* Mountain View, Ca. (1985).

[9] R. Levin. *Supporting a programming environment on a multiprocessor workstation,* First Bay Area Systems Seminar (1985).

[10] R. D. Gaglianello and H. P. Katseff. *Meglos: An operating system for a multiprocessor environment,* AT&T Bell Laboratories, Holmdel, N.J. (1985).

[11] H. C. Lauer and R. M. Needham, "On the duality of operating system structures," *Operating Systems Review 13,* 4, (1979).

Interprocess Communication in the Eighth Edition Unix System

D. L. Presotto
D. M. Ritchie

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

When processes wish to communicate, they must first establish communication, and then decide what to say. Previously described stream mechanisms of the Eighth Edition Unix system[1] provide a flexible way for processes to speak with devices and with each other. An existing stream connection is named by a file descriptor, and the usual read, write, and I/O control requests apply. Processing modules may be inserted dynamically into a stream connection, so network protocols, terminal processing, and device drivers separate cleanly.

This paper describes ways of commencing communication. The traditional, and simplest, is the pipe. In our system, pipes are just cross-connected streams.

A new request associates a stream with any named file. When the file is subsequently opened, operations on the file are operations on the stream. Also, open files may be passed from one process to another over a stream, and opening a stream-associated file may create a new and distinct channel to the stream's server.

These low-level mechanisms allow considerable flexibility in constructing network dialout routines and connection servers of various kinds.

Introduction

The Eighth Edition version of Unix is the system that runs on machines in the Information Sciences Research Division of AT&T Bell Laboratories, and at a few sites elsewhere. It is named, by our custom, after its manual.

The work reported here provides convenient ways for programs to establish communication with unrelated processes, on the same or different machines. The kind of communication we are interested in is conducted by means of ordinary read and write calls, occasionally supplemented by I/O control requests. Moreover, we wish to commence communication in ways that are as close as possible to ordinary file opening. These considerations spring from the desire to find ways of fitting even complicated things into a simple pattern, and from the observation that whenever objects behave like files, practically any program is able to use them.

In particular, we study how to

1) provide objects nameable as files that invoke useful services, such as connecting to other machines over various media,
2) make it easy to write the programs that provide the services.

Unix is a trademark of AT&T Bell Laboratories

Recapitulation

The Eighth Edition system introduced a new way of communicating with terminal and network devices,[1] and a generalization of the internal interface to the file system.[2,3] We begin by reviewing already-published nomenclature and mechanisms of our I/O and file systems.

Streams

A *stream* is a full-duplex connection between a process and a device or another process. It consists of several linearly connected processing modules, and is analogous to a Shell pipeline, except that data flows in both directions. The modules in a stream communicate by passing messages to their neighbors. A module provides only one entry point to each neighbor, namely a routine that accepts messages.

At the end of the stream closest to the process is a set of routines that provide the interface to the rest of the system. A user's *write* and I/O control requests are turned into messages sent along the stream, and *read* requests take data from the stream and pass it to the user. At the other end of the stream is either a device driver module, or another process. Data arriving on the stream is sent to the device or read by the process; data and state transitions detected by the device, or generated by the process, are composed into messages and sent into the stream towards the user program. Intermediate modules process the messages in various ways. They are symmetrical; their read and write interfaces are identical.

The two end modules in a stream to a device become connected automatically when the process opens the device; streams between processes are created by a *pipe* call. Intermediate modules are attached dynamically by request of the user's program. They are addressed like a stack with its top close to the process, so installing one is called 'pushing' a new module.

Queues

Each stream processing module consists of a pair of *queues,* one for each direction. A queue comprises not only a data queue proper, but also two routines and some status information. One routine is the *put procedure,* which is called by its neighbor to place messages on its data queue. The other, the *service procedure,* is scheduled to execute whenever there is work for it to do. The status information includes a pointer to the next queue downstream, various flags, and a pointer to additional state information required by the instantiation of the queue. Queues are allocated in such a way that the routines associated with one half of a stream module may find the queue associated with the other half. (This is used, for example, in generating echoes for terminal input.)

Message blocks

The objects passed between queues are blocks obtained from an allocator. Each contains a *read pointer,* a *write pointer,* and a *limit pointer,* which specify respectively the beginning of information being passed, its end, and a bound on the extent to which the write pointer may be increased.

The header of a block specifies its type; the most common blocks contain data. Control blocks of several kinds have the same form as data blocks and are obtained from the same allocator. For example, there are control blocks to introduce delimiters into the data stream, to pass user I/O control requests, and to announce special conditions such as line break and carrier loss on terminal devices.

Examples

Figure 1 shows a stream device that has just been opened. The top-level routines, drawn as a pair of half-open rectangles on the left, are invoked by users' *read* and *write* calls. The writer routine sends messages to the device driver shown on the right. Data arriving from the device is composed into messages sent to the top-level reader routine, which returns the data to the user process when it executes *read*.

Figure 2 shows an ordinary terminal connected by an RS-232 line. Here a processing module

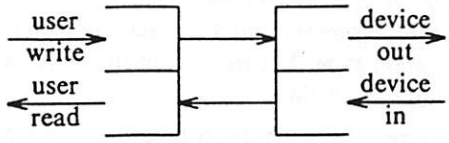


Figure 1. Configuration after device open.

(the pair of rectangles in the middle) is interposed; it performs the services necessary to make terminals usable, for example echoing, character-erase and line-kill, tab expansion as required, and translation between carriage-return and new-line.

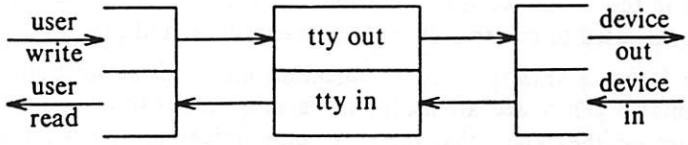


Figure 2. Configuration for normal terminal attachment.

Many network connections require flow- and error-control protocols to be carried out by the host computer. Therefore, when terminals are connected to a host through such a network, a setup like that shown in Fig. 3 is used; the terminal processing module is stacked on the network protocol module.

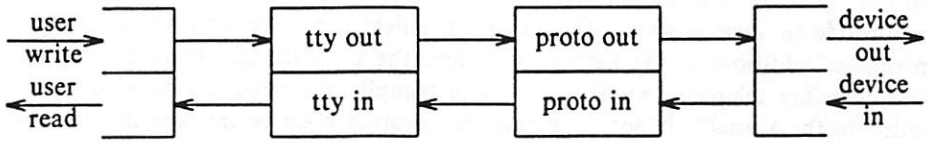


Figure 3. Configuration for network terminals.

A common fourth configuration (not illustrated) is used when the network is used for file transfers or other purposes when terminal processing is not needed. It simply omits the "tty" module and uses only the protocol module. Sometimes, on the other hand, a front-end processor conducts the required network protocol. Here a connection for remote file transfer will resemble that of Fig. 1, because the protocol is handled outside the operating system; likewise network terminal connections via the front end may be handled as shown in Fig. 2.

Finally, Figure 4 shows the connections for a pipe. In our system, pipes are full-duplex.

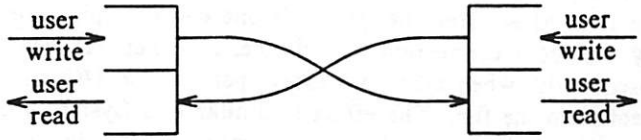


Figure 4. A pipe.

File Systems

Weinberger[2] generalized the file system. He identified a small set of primitive operations on inodes (read, write, get, put, truncate, get status, etc.: a total of 11) and created a form of the *mount* request that specifies a file system type and, where appropriate, a stream. When file

operations are requested, the calls to the underlying primitives are routed through a switch table indexed by the type. His file system type performs remote procedure calls across the associated stream, at the other end of which is a server, and thus accomplishes a remote file system. Pike[4] takes advantage of the same file system type, but his server simulates a disk containing images classified by machine, person's name, and resolution.

Killian[3] added a file system type that appears to be a directory containing the names (process ID numbers) of currently running processes. Once a process file is opened, its memory may be read or written, and control operations can start it or stop it.

Establishing communication

Traditional Unix systems provide few ways for a process to establish communication with another. The oldest one, the pipe, has proved astonishingly valuable despite its limitations, and indeed remains central in the design we shall describe. Its cardinal limitation is, of course, that it is anonymous, and cannot be used to create a channel between unrelated processes.

AT&T's System V has a variety of communication mechanisms including semaphores, messages, and shared memory. They are all useful in certain circumstances, but programs that use them are all special-purpose; they know that they are communicating over a certain kind of channel, and must use special calls and techniques. System V also provides named pipes (FIFOs). They reside in the file system, and ordinary I/O operations apply to them. They can provide a convenient place for processes to meet. However, because the messages of all writers are intermingled, writers must observe a carefully designed, application-specific protocol when using them. Moreover, FIFOs supply only one-way communication; to receive a reply from a process reached through a FIFO, it is necessary to construct the return channel explicitly.

Berkeley's 4.2 BSD system introduced sockets (communication connection points) that exist in domains (naming spaces). The design is powerful enough to provide most of the needed facilities, but is uncomfortable in various ways. For example, unless extensive libraries are used, creating a new domain implies additions to the kernel. Consider the problem of adding a 'phone' domain, in which the addresses are telephone numbers. Should complicated negotiations with various kinds of ACUs be added to the kernel? If not, how can the required code be invoked in user mode when a program calls 4.2's *connect* primitive?

The Eighth Edition's variant file systems, mentioned above, provide a general way of establishing communications between processes. Indeed, they are a bit too general for many problems; it requires a sophisticated server to simulate a file system. Therefore, we tried to find simpler, but still general, ways of connecting programs.

Additions

We made three additions to the system.

Mounted streams

First is a new, but very simple, file system type. Its *mount* request attaches a stream named by a file descriptor to a file. Most often the stream is one end of a pipe created by the server process, but it can equally well be a connection to a device, or a network connection to a process on another machine. Subsequently, when other processes open and do I/O on that file, their requests refer to the stream attached to the file. The effect is similar to a System V FIFO that has already been opened by a server, but more general: communication is full-duplex, the server can be on another machine, and (because the connection is a stream), intermediate processing modules may be installed.

By itself, a mounted stream shares the most important difficulty of the FIFO; several processes attempting to use it simultaneously must somehow cooperate.

Passing files

The second addition is a way of passing an open file from one process to another across a pipe connection. Although they are actually done with *ioctl* operations, the primitives may be written

```
sendfile(wpipefd, fd);
```

in the sender process, and

```
(fd1, info) = recvfile(rpipefd);
```

in the receiver. The sender transmits a copy of its file descriptor *fd* over the pipe to the receiver; when the receiver accepts it, it gains a new open file denoted by *fd1*. (Other information, such as the user- and group-id of the sender, is also passed.)

Unique connections

Finally, we found a way for each client of a server to gain a unique, non-multiplexed connection to that server. It takes the form of a processing module that can be pushed on a stream, which will usually be mounted in the file system as described above. When the file is opened by another program, this module creates a new pipe, and sends one end to the server process at the other end of the mounted stream, using the same mechanism as the *sendfile* primitive described above. After the server has called *recvfile* to pick up its end of the pipe, it may accept or reject the new connection; if it accepts, the other program's *open* call succeeds, and its open file refers to the local end of the new pipe to the server. If the server rejects the request, the *open* fails.

Examples

A graded set of examples will illustrate how to use these facilities.

Network calling

Originating network connections is a complicated activity. There is often name translation of various kinds, and sometimes negotiations with various entities. With our Datakit VCS network,[5] for example, a call is placed by negotiating with a node controller. When dialing over the switched telephone system, one must talk to any of several kinds of ACU devices. Setting up a connection on an Ethernet under any of the extant protocols requires translation of a symbolic name to a net address. These protocols should certainly not be in kernel code. It is usual to put setup negotiations in user-callable libraries, but it is better to have all the code for each network in a single executable file. In this way, if something in the network interface changes, only one program needs to be fixed and reinstalled.

A program desiring to make a connection calls a simple routine that creates a pipe, forks, and in the child process executes the network dialer process. The dialer either returns an error code, or passes back a file descriptor referring to an open connection to the other machine. The pseudo-code for the library routine, neglecting error-checking and closing down the pipe, is:

```
netcall(address)
{       int p[2];
        pipe(p);
        if (fork()!=0)
                execute("/etc/netcaller", address, ascii(p[0]));
        status = wait();
        if (bad(status))
                return(errcode);
        passedfd = recvfile(p[1]);
        return(passedfd.fd);
}
```

The `/etc/netcaller` program can be arbitrarily complicated. Its job is to create the connection and either fail, returning an appropriate error code, or succeed, and pass its descriptor for the open

connection. Along the way, it may negotiate permissions and provide the caller's identity reliably, because it can be a privileged (set-uid) program.

Process connections

Suppose you are writing a multi-player game, in which several people interact with each other through a controller process. It might be a banker (Monopoly) or a mazekeeper (Mazewar) or a tournament director (Bridge). The problem is to set up a single process prepared to receive asynchronous connection requests from new players. In our solution, there are two programs: the controller, set up initially, and the player program, executed by users as they enter the game. When the controller starts, it creates a pipe, pushes the unique-connection processor on one end of the pipe, mounts it on a conventionally-named file (say `/tmp/gamester`), and waits for connection messages to arrive. When the player program is run, it opens `/tmp/gamester`, thereby doing an implicit *sendfile*. The controller notices that there is input on its connection pipe (perhaps making use of *select*) and accepts the connection with *recvfile*. Thereafter, the player program transmits moves and receives replies over the file descriptor obtained from opening the gamester file, and the controller reads the player's moves and transmits replies over the file descriptor it received.

Network calling (advanced course)

The network calling routines described above can be generalized by writing a connection server. The mechanism is illustrated in Figure 5. In structure, the connection server is exactly like the game master; it maintains a conventional file (say `/server`) with a stream mounted on it (5a).

Users of the service (programs like *cu* and *rlogin*, or our local addition to the family, *dcon*), open `/server` and write on it the name of the entity they wish to contact (5b). The server program reads the name, and translates it to a true address and the name of one of the network caller programs described above. It then executes the caller program (5c) and passes the resulting connection file back to *dcon* (5d). Finally, *dcon* accepts the new connection and closes the connection stream (5e).

A fine point in the design is the decision to make the connection server a continuously-existing process, and to communicate the desired address by writing on its mounted file. Another possibility is to invoke it like the caller programs in the first example: by executing it, with the user's address as a parameter. From the user's viewpoint, these two interfaces is equally effective. We are trying the current approach because it seems more efficient, in that the mapping tables can be cached in the server process, and it requires fewer process creations. It is also a more interesting experiment in program design, because it is so highly multiplexed.

The structure encourages experimentation with naming plans. A domain setup seems natural to try. Suppose we have domains *dk*, *inet*, and *att* that refer respectively to Datakit, Internet, and the switched telephone system. Then the following address translations, which yield a complete Datakit address, Internet host address, and telephone number, are appropriate for our gateway machine:

```
dk/research      →    mh/astro/research
inet/research    →    192.11.4.55
att/research     →    2015825940
```

There are, of course, complications: the machine is on two ethernets that have to be distinguished; various line speeds have to be specified when dialing with an ACU. Finally, it is necessary to coalesce the *dcon*, *rlogin*, and *cu* programs.

Conclusion

Unix has always had a rich file system structure, both in its naming scheme (hierarchical directories) and in the properties of open files (disk files, devices, pipes). The Eighth Edition exploits the file system even more insistently than its predecessors or contemporaries of the same genus. Remote file systems, process files, and the face server all create objects, the name of which can be handed as usefully to an existing tool as to a new one designed to take advantage of the object's special properties. Similarly, the stream I/O system provides a framework for making

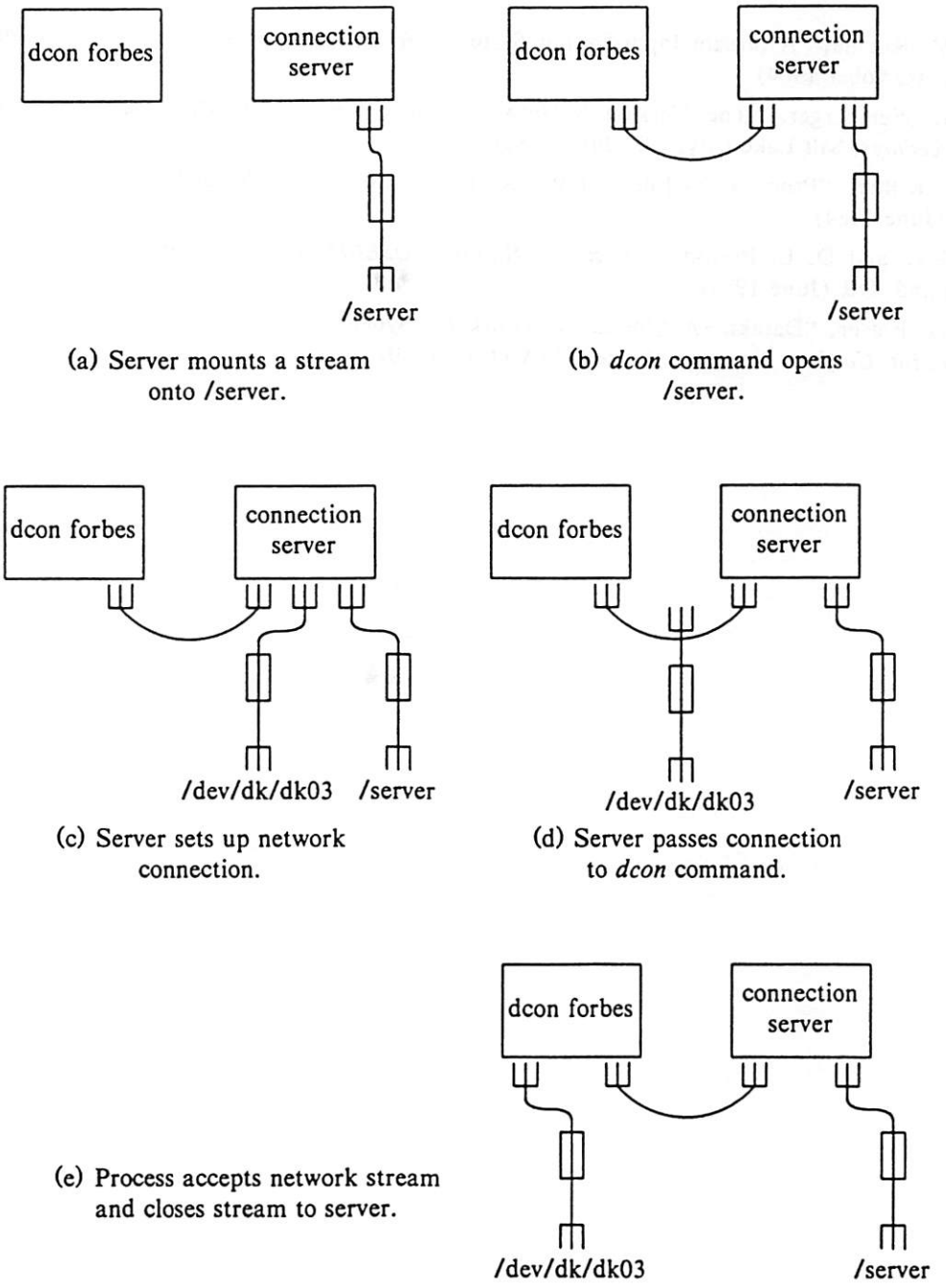


Figure 5. Establishing a network connection.

already opened files behave in the standard way most programs already expect. For example, the real purpose of the terminal-processing module is to mediate between programs expecting a simple byte stream, and imperfect typists using terminals with peculiar control requirements. A network protocol module's purpose is to make an error-prone network, again with idiosyncratic properties, conform to a simpler model.

The developments described here follow the same path; they encourage use of the file name space to establish communication between processes. In the best of cases, merely opening a named file is enough. More complicated situations require more involved negotiations, but the file system still supplies the point of contact. Moreover, the necessary negotiations may be encapsulated in a common form that hides the differences between local and any of a variety of remote connections.

References

1. D. M. Ritchie, "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal* **63**(8) (October 1984).
2. P. J. Weinberger, "The Version 8 Network File System," *USENIX Summer Conference Proceedings*, Salt Lake City, UT (June 1984).
3. T. J. Killian, "Processes as Files," *USENIX Summer Conference Proceedings*, Salt Lake City, UT (June 1984).
4. R. Pike and D. L. Presotto, "Face the Nation," *USENIX Summer Conference Proceedings*, Portland, OR (June 1985).
5. A. G. Fraser, "Datakit—A Modular Network for Synchronous and Asynchronous Traffic," *Proc. Int. Conf. on Commun.*, Boston, MA (June 1980).

User-Level Shared Variables (in a Hierarchical Control Environment)

Don Libes

National Bureau of Standards
Factory Automation Systems Division
Gaithersburg, MD 20899

ABSTRACT

We have implemented a shared variable system for UNIX† 4.2BSD [Joy] that emulates a shared or common memory.

- This system is all user level code and requires no kernel modifications. It is accessible from a variety of languages.
- This shared memory system provides its service transparently to remote hosts across a local-area network.
- This system is being used in a real application - an automated factory. Shared variables are an appropriate communication paradigm in a real-time environment (in comparison to message passing or pipes).

Background

The National Bureau of Standards has been performing research in the area of robot control [Albus] for over a decade. Robot control is typified by tightly-coupled multiprocessor systems often implemented as multiple single-board computers residing in a common bus. These systems require frequent and fast transfer of small amounts of data.

For example, a robot arm may require new joint angles 100 times a second in order to move smoothly. These joint angles are very small pieces of data and further, the number of joints and joint angles is constant, hence they can be stored in "well-known" locations.

To support this type of application, multiported memory is used so that several different computers can access the same locations. In the robot example, the controlling computer and the servo computers each need access to the information.

Common memory has the further advantage that as new processes are added that need information already present, extant processes do not have to be modified to deliver that information. For example, a process was added that displays the robot's actions on a graphics monitor. The display process was added without modification to any other part of the system, since it uses the joint angles which are stored in well-known common memory locations. Recently a "safety" process was added to guarantee that the robot never departs its working envelope. The safety process also obtains its information from common memory.

In 1981, work [Furlani] began on an automated factory. This is an extension of the robot control system in many ways. It is expected that the factory model will grow over time, and become richer in the number of processes used and the amount of information shared between processes. A major difference in the automated factory project is that systems which have to

† UNIX is a Trademark of Bell Laboratories.

communicate are often in separate backplanes and use different operating systems. Common memory is seen as a consistent communications methodology for this disparate collection of computers.

Today, as computers grow larger and become more complex, more and more layers of protocol prevent ready access to common memory. Indeed, we find that in most of our computer-to-computer interfaces, common memory is now simply a way of thinking, rather than an underlying implementation. Such is the case with UNIX 4.2BSD.

The following section describes an implementation of the common memory paradigm in the UNIX 4.2BSD environment.

I. User View

A server, the Common Memory Manager (CMM), acts on requests to access shared variables. Typical requests are read, write, and declare. Variables are structured and have attributes such as type, protection, lifetime, and ownership. Variable type checking is performed at runtime startup, since processes are loosely connected.

Some of the more interesting attributes of these shared variables are:

lifetime and "shelf life". In a real time environment, the system must survive a process dying or getting bogged down temporarily. When the useful lifetime of a variable's value has expired, other processes are free to manipulate the variable, for instance, by redeclaring or writing it. This attribute allows a "god" process to notice unexpected process deaths and restart processes or transfer control to other processes. For less critical variables (such as an infrequently updated sensor value), processes can note that a value is "stale", and make a projection that will carry them over until the variable writer catches up to its duties.

write count, timestamp, authorstamp. By marking each variable with how many times or when it has been written, handshaking can be performed between processes. The archetypal example from the automation environment has a supervisor sending the command "hit nail with hammer" to a subordinate. If the supervisor runs more quickly than the subordinate, without handshaking, the subordinate may never see the command (i.e. nail is not hit). If the subordinate runs more quickly than the supervisor, without handshaking, the subordinate may execute the same command twice (i.e. nail is hit more than once).

The same problem exists when the supervisor reads the feedback from the subordinate. In particular, because there is no synchrony between processes, the supervisor has no way of knowing what command the subordinate is responding to (without handshaking). Actual code for such a sequence would use higher-level routines to read the shared variable. For example:

```
struct shared_variable *command_out, *feedback_in;
struct shared_variable_value *expected_feedback;
.
.
if (status_equal(command_out, feedback_in, expected_feedback)) . . .
```

Here, the command to the subordinate is **command_out**. The actual feedback received is **feedback_in** and we are looking for **expected_feedback**. The expected feedback is thus compared to the feedback actually received with respect to the command which generated it.

These handshaking and synchronization problems are thoroughly discussed in [Libes].

wakeup. Variables can be marked with a list of processes to be notified upon value change. This was originally added for efficiency, so that clients, for example, would not have to loop just waiting for new values. It has turned out to be extremely useful for debugging. Rather than inserting print statements in existing code and recompiling, it is possible to create a debug module that simply prints out variables when they change. It has also been used for a graphics monitor

that continuously displays the state of the entire system.

The wakeup attribute can be used to force a queuing discipline on a variable. For example, a process performing as a resource server may take requests from a well-known common memory variable. As clients write to the variable, the CMM wakes up the resource server with each new variable value. The resource server typically responds to each variable value in turn; however it can ignore all but the most recent values if it desires.

non-exclusive write. Variables can be declared read or write, exclusively or nonexclusively. By default, writes are exclusive, meaning that only one process can write a variable during its lifetime. Non-exclusive write has no such restriction, allowing multiple processes to write the same variable. One use of this might be a server listening at a well-known location ("socket") for a request for service. Any client can request service by writing the variable associated with the socket.

Another use is to send information to a log file. In our application, we can ask all processes to log their states a given number of clock ticks. By having them all write their states to one variable, a logging process can record all the states of the system through time. This has proven to be invaluable during system testing.

II. 4.2BSD Implementation

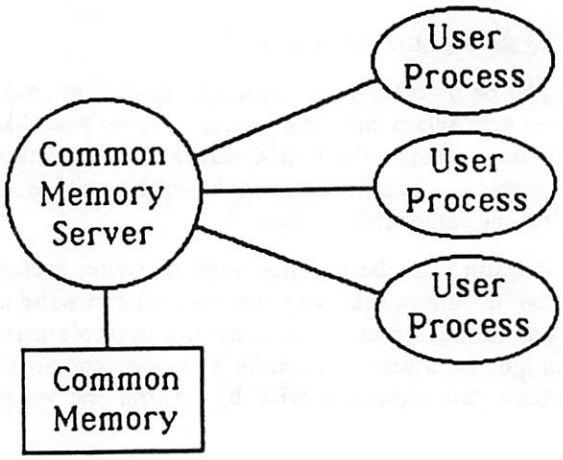
The Shared Variables System (SVS) is written in C and uses the 4.2 interprocess communication system [Leffler]. An interface exists for Franz Lisp.

The SVS consists of three layers. The lowest level simulates a common memory facility such as is available in System V. On top of this are the procedures that transform common memory into common variables. This level provides users with routines allowing access and control of the common variables.

Specific to our use of the SVS is a higher level that customizes it for communication in a hierarchical control environment. This level includes interprocess handshaking.

Building a server on 4.2

4.2BSD has no common memory facility. Our system provides one via a server that stores variables in its own memory space which is private to it. (See Figure 1) The server runs as a user-level process which needs no special privileges and can be run from an unrelated user-id.

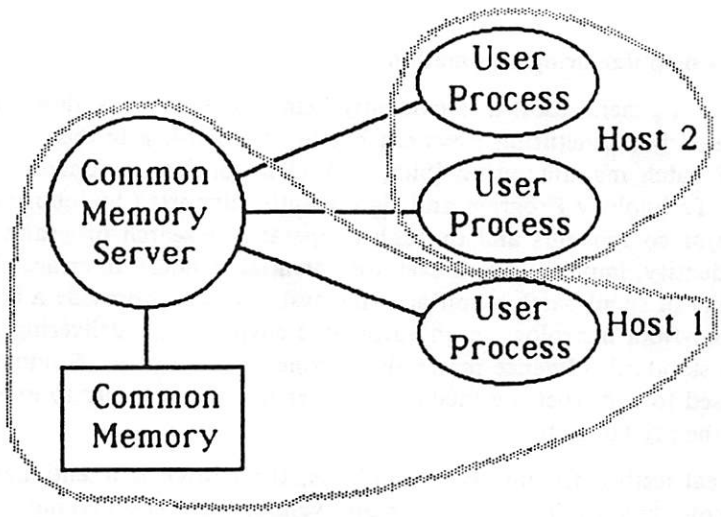


Shared Variable System
Figure 1

Communications between the common memory server and clients have properties of both streams and datagrams. Communication resembles virtual circuits in that the server and clients pass many messages between them in a conversation (like TCP). These messages are also record-oriented (like UDP). In implementation, UDP was rejected because it would have forced us to handle unreliability and fragmentation (datagrams were limited to 1K which was smaller than the typical message).

In this case, we chose TCP as the lesser of two evils. TCP had the drawback (for us) of removing record boundaries (i.e. the receiver could not tell where one write ended and the next began). To handle this, new versions of read and write were constructed to do the actual I/O to the sockets. These packetized and depacketized the TCP streams back into records.

Since TCP/IP is capable of communicating across the network, clients may be distributed (figure 2) while accessing the common memory. However, the small number of concurrent TCP connections (viz. the number of open files) is a severe limit. Another system [Mitchell] glues distributed common memories together throughout the factory. This design allows multiple common memories to exist but appear to the user as one. This gluing process bypasses the TCP restriction and allows multiple instances of the 4.2 (as well as others') common memory service to be run.



Distributed Clients
Figure 2

Simulating common memory on 4.2

The server's own private memory serves as common memory. Access to variables, therefore, is strictly through the valid requests to the CMM server. The server accesses variables as requested and maintains extra information such as variable type, owner, etc.

Whenever a client wants to access common memory, it does so by sending a message to the CMM. Such messages are sent asynchronously but received synchronously. In particular, the server is never interrupted.

Typically, the server is waiting for service requests. When a request is received, service is performed. This service may or may not cause sending of a message in return to the client. For example, "read" operations always cause information to be returned from the server while "write" operations don't (unless an error is encountered).

It is up to the client whether it wishes to wait (even after a "read") for a response from the common memory server.

Client processes keep their own local version of shared variables of interest until it is convenient for them to synchronize with the CMM. By calling **cm_sync()**, the user and the common memory server become synchronized. An argument to **cm_sync()** specifies whether the client waits for a response or not.

Whether the client is waiting or not, an error can occur at the server end, in which case an error message is returned. For example, a client may write a value but the variable may be an incompatible type. When this happens, a message is sent back to the client. The client is not interrupted, but at the next **cm_sync()** operation it is notified of the error.

Interestingly, the problem of being able to provide atomic updates to common memory vanishes in this system. Since the server is never interrupted, client updates functionally prevent all other clients access to common memory while their request is being serviced. This removes the classic synchronization problem which exists when multiple processes are reading and writing real common memory with no synchronization. On the other hand, this can cause relatively long waits for service.

Construction of higher-level synchronization such as semaphores is easy. Potentially stalling manipulations of semaphores and monitors can call **cm_sync()** and then wait for a response from the server to continue. The server acknowledges the client if and only if the client

semaphore action is satisfied.

III. An application - manufacturing automation

While the SVS is a general tool, it is currently being used at the National Bureau of Standards in the Automated Manufacturing Research Facility (AMRF), a testbed for research in the automation of small batch manufacturing [Simpson]. The project is funded by NBS and the Navy Manufacturing Technology Program and significantly supported by industry through donations or loans of major components and through cooperative research programs. The objective of the project is to identify, implement and exercise potential standard interfaces between existing and future components of small-batch manufacturing systems and to provide a laboratory for the development of factory-floor metrology in an automated environment, delivering proven measurement techniques and standard reference materials to American industry. Commercially available products are being used to construct the facility wherever possible in order to expedite transfer of research results into the private sector.

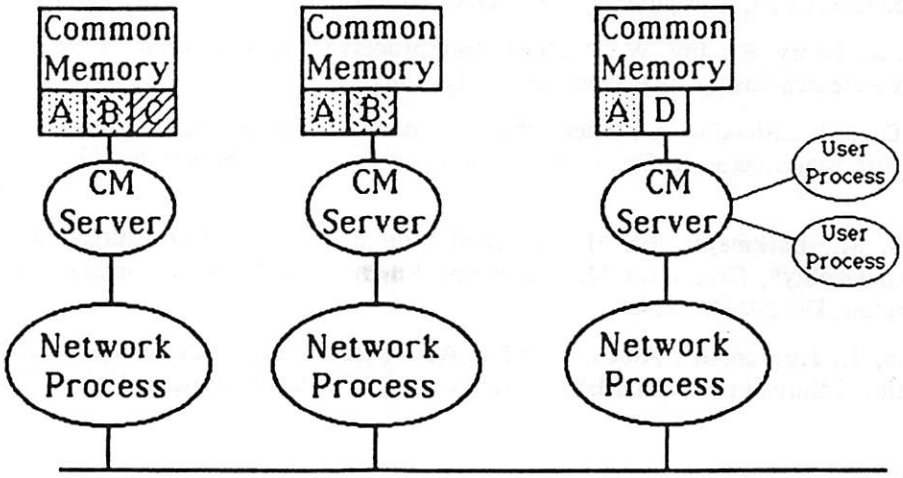
To provide a real testbed for interface standards, the AMRF is intentionally composed of manufacturing and computer equipment from many vendors, thereby making its construction a major integration effort. The types of systems that must be integrated and communicate with each other include stationary and mobile robots, machine tools, their controllers, and higher-level cell and factory management systems. Actual computers and operating systems run the gamut, from micros to mainframes.

Processes in the AMRF are organized in a control hierarchy, much like the organization of a conventional factory. At each level, tasks are broken down into simpler tasks that can be performed by subordinate processes. Control is hierarchical.

To support such control communication, we imagine that processes in the system communicate along hierarchical channels. Each process runs asynchronously, continually "sending" commands to subordinate processes and feedback to superior processes.

Logically however, communication is performed using the common memory model. For example, a process sending commands to a subordinate writes the information in a well-known common memory variable. The subordinate reads the information out of the common memory variable, noting whether or not it contains a new value.

In reality, there are small groups of processes and processors that directly communicate with each other, some using physical common memory and some simulating common memory. Each cluster representing a piece of common memory can choose to replicate any other portion of another cluster's common memory. An underlying network (figure 3) invisibly supplies this service of keeping all the common memories synchronized. The result is effectively one of a consolidated common memory, with a very consistent and easy means of communication between processes.



Network glues Common Memories together

Figure 3

Summary

Common memory is a communications methodology that is easy to use and can be efficiently implemented. It can use physical common memory or be simulated by other communications techniques such as message passing. In turn, it can be used to implement any other communications techniques.

Common memory has several important advantages and disadvantages. The primary advantage is the ease with which information may be shared. In our application, we have required this richness of data sharing between processes. Using common memory also has the benefit of being able to add, remove or modify processes without disturbing other processes, *even if they communicate with each other.*

The primary disadvantage is that sharing large amounts of information in this manner can become hard to keep track of. Since there is no scoping in the common memory, naming conventions must be followed (which in effect, provides scoping). Common memory is not particularly apt for all types of communications. Picking the most appropriate model (pipes, databases, message passing) will invariably lead to a simpler process in the final analysis. We, of course, use these other techniques in our application when appropriate.

At the lowest level of our application, efficient process-to-process communication is necessary for real-time behavior. This is implemented using real common memory with low overhead. At higher levels where common memory is simulated and the high-speed demand does not exist, the methodology extends upwards suitably.

In the future, we expect to move the ideas of hierarchical control and common memory to new applications. Currently, we are investigating the potential for control of autonomous aircraft.

References

1. Albus, J., Barbera, T., Fitzgerald, M.L., "Hierarchical Control for Sensory Interactive Robots", Proceedings of 31st General Assembly, International Institution for Production Engineering Research (CIRP), Toronto, Canada, September, 1981.

2. Furlani, C., Kent, E., Bloom, H., McLean, C., "The Automated Manufacturing Research Facility of the National Bureau of Standards", Proceedings of the Summer Simulation Conference, Vancouver, B.C., Canada, July 1983.

3. Joy, W., Cooper, E., Fabry, R., Leffler, S., McKusick, K., Mosher, D., "4.2BSD System Interface Overview", Computer Systems Research Group, U.C. Berkeley, July, 1983.

4. Leffler, S., Fabry, R., Joy, W., "4.2BSD Interprocess Communications Primer", Computer Systems Research Group, U.C. Berkeley, July, 1983.

5. Libes, D., "Handshaking in a Hierarchical Control Environment", Internal Memorandum, Center for Manufacturing Engineering, National Bureau of Standards, Washington, DC 20234.

6. Mitchell, M., Barkmeyer, E., "Data Distribution in the NBS Automated Manufacturing Research Facility", Center for Manufacturing Engineering, National Bureau of Standards, Washington, DC 20234, 1984.

7. Simpson, J., Hocken, R., Albus, J., "The Automated Manufacturing Research Facility of the National Bureau of Standards", Journal of Manufacturing Systems, Vol. 1, #1.

Further Reading

1. Bloom, H., Furlani, C., Barbera, A., "Emulation as a Design Tool in the Development of Real-Time Control Systems", 1984 Winter Simulation Conference, Dallas, Texas, November 1984.

2. Johnson, T., Milligan, S., Fortmann, T., Bloom, H., McLean, C., Furlani, C. "Emulation/Simulation of a Modular Hierarchical Feedback System", The 21st IEEE Conference on Decision and Control, Orlando, FL, December 1982.

LEVI: A Prototype Active Assistance Interface

Manton M. Matthews
and
Ted Nolan

Department of Computer Science
University of South Carolina, Columbia, S.C. 29208
ihnp4!akgua!usceast!matthews
ihnp4!akgua!usceast!ted

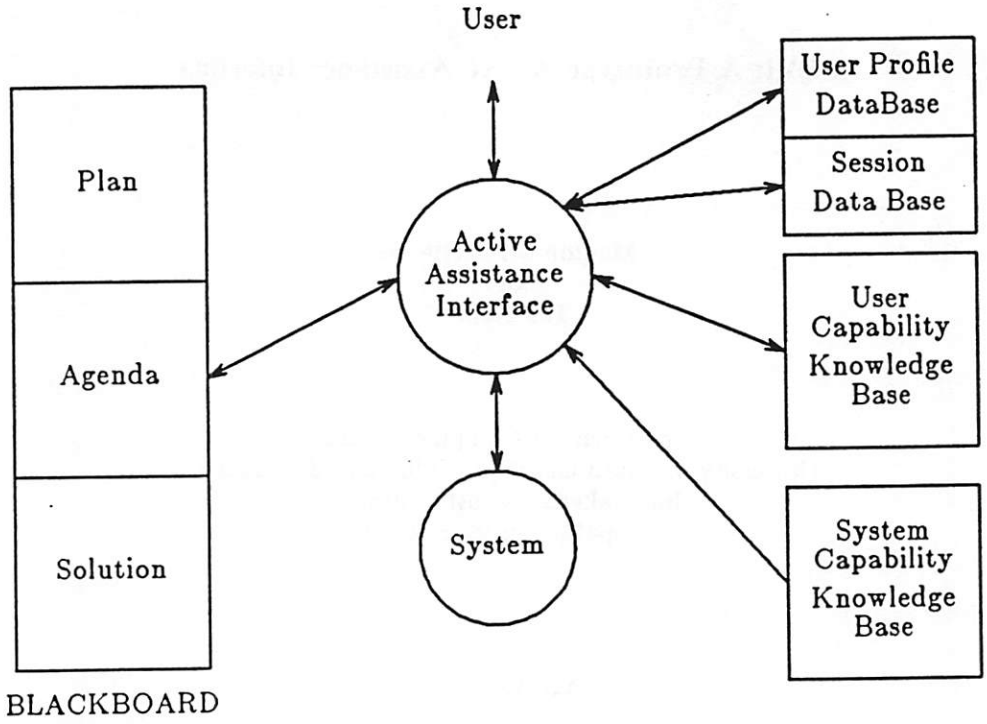
ABSTRACT

In this paper we discuss Levi (Learning Extended vi) a prototype active assistance system that guides the user in learning the capabilities of the screen editor vi. An active intelligent assistant interface operates in many ways like a human expert constantly observing the user-system interaction. On occasion the interface will, as the human expert would, observe things that the user could have done in a more effective manner. When this occurs the interface will, taking into consideration the level of experience of the user, suggest how the task might be accomplished more efficiently. An active assistance interface has two major components: an active assistant and an intelligent assistant. The intelligent component of the facility allows a system. Responses are tailored to the user's proficiency in the use of the software system and current session characteristics. The active component monitors user interactions with the system to "learn" his proficiency and determine current session characteristics. Using this information, it makes decisions about when to prompt a user and suggest more efficient methods for accomplishing certain tasks. The aim is to build assistance systems that not only provide users an easy access to system capabilities, but also provide a mechanism that enables them to gradually learn and become experts in the use of the software system.

1. Introduction

Since the advent of on-line manuals there has been considerable research directed towards improving the capabilities of online assistance systems. This includes work on natural language interfaces [14, 16, 22, 32, 33], interfaces that model users and user intentions [15, 21], intelligent assistants [5, 7, 8, 11, 22, 31], and intelligent on-line monitors [9, 28]. All of these systems with the exception of the on-line monitors are passive in nature. They respond only to direct queries from the user. An active system is one that on occasion initiates the interaction by making suggestions to the user. The work in on-line monitors by Griesmer et al, has a system monitoring the operation of a computer center and making suggestions to the operator. Our system is similar at a very high level in that it monitors the user-system interaction and, using knowledge of the capabilities of the user and the system as well as characteristics about the particular session, makes suggestions as to the best course of action.

The goal of our research project is to develop an independent active assistance interface as shown in Figure 1. This interface will fit between the user and any software system. The interface will monitor commands and maintain knowledge on the user's expertise with the



ACTIVE ASSISTANCE INTERFACE

FIGURE 1

software system as the commands are passed through to the underlying system. Using this information and at appropriate times the system makes suggestions about how the user can more effectively use the system. The appropriate times to make suggestions fall into three categories:

(1) after the user has made an error or sequence of errors,

(2) when an inefficient sequence of commands is recognized, and

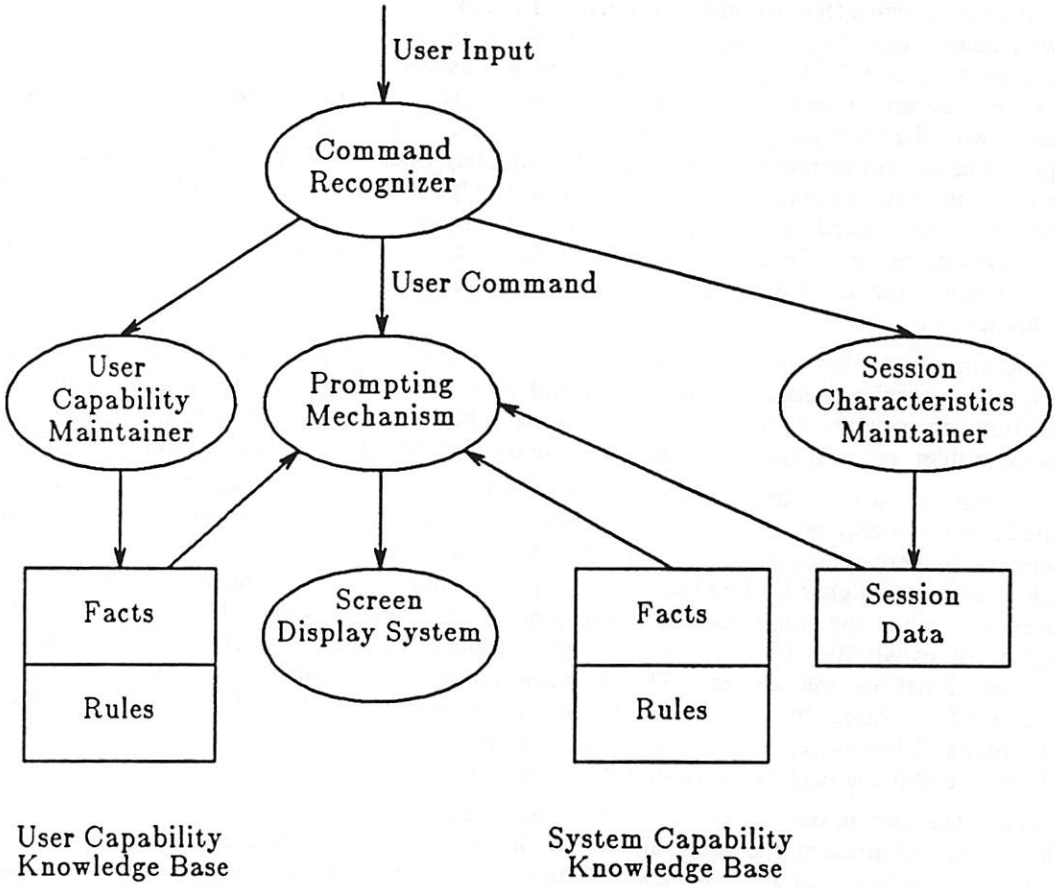
(3) periodically to guide the user in learning the capabilities of the system, filling in gaps in the user's knowledge of such capabilities.

In some cases, in particular when errors occur, the interface will be able to gain useful information from the responses of the system to the user, but in most cases these responses will be passed directly through. While the interface will be independent of the particular system, it will maintain a separate knowledge base for each user and software system. The blackboard, shown in Figure 1, is a standard component of a knowledge based system where intermediate goals are written [10]. Within the Active Assistance Interface, there is an inference engine that will take goals from the blackboard and decompose them into simpler goals. This process (forward chaining) is repeated until the final solution is obtained.

In this research, a knowledge-based approach is utilized in constructing an active oniine assistance system [2, 16]. The knowledge base of this system can be subdivided into two major components. The first contains information about the capabilities of the software system and methods for its use, and the second maintains general rules and procedures for determining user capabilities and session characteristics. The assistance system also needs to maintain a separate

database for each user to accumulate information on the user's capabilities in the use of various aspects of the system. In addition, the database has a component that records current session characteristics. Our research differs from earlier work in that our system is active and in the extent of the models of the user and system.

Our model of the major components of the general active assistance interface is shown in Figure 2. The system maintains three separate classes of data: knowledge of user capabilities, knowledge of system capabilities, and session characteristic data. The knowledge of the system is static; the system does not change in capability as the user interacts with it. If there are modifications to a system, then effectively a new system with a new system capability knowledge base is installed. The major modules of the active assistance interface are: the command recognizer, the prompting mechanism, the screen display system and modules for maintaining user capabilities and session characteristics.



MAJOR COMPONENTS
OF AN ACTIVE ASSISTANCE INTERFACE

FIGURE 2

The command recognizer uses knowledge of the system to separate commands from other incoming text, which is merely passed directly through. The recognized commands are sent to the routines that maintain the user capabilities and session characteristics. The prompting mechanism then, using the information in the user and system capability knowledge bases and the session characteristics, decides when to become active and prompt the user. It also decides on what particular aspect of the system the user should be prompted.

2. LEVI: An Initial Prototype Active Assistance Interface

The user interaction with an active assistance system is radically different than with a passive system. When we began this research we had no way to judge how users would react to this different mode of interaction. To enable us to get feedback from users as soon as possible, and give us experience in designing an active interface, we developed a prototype active assistance system. This prototype was an active assistance module for vi (LEVI). Vi is a screen editor that runs under the UNIX (a trademark of AT&T Bell Laboratories) operating system [13]. The complexity of the editor encourages users to reach a point where they have learned enough to use the system, but not to attempt to progress further and become more proficient by learning how to utilize some of the more powerful and beneficial features. LEVI maintains in the file ".viknowledge" information on the user's capability with vi. Periodically when the user initiates a vi session, the system offers to inform the user of a capability of vi that the user has not used, or otherwise demonstrated knowledge of. The length of time between prompts varies depending on information such as how long the user's account has existed, how much knowledge the user has about vi, and an annoyance factor. The annoyance factor is an attempt to measure how annoyed the user is with the prompting. This is based on the percentage of affirmative answers to previous prompts. The system prompts a new user approximately once a day (not every time the user uses vi) and as the user becomes more experienced the period between promptings grows. This prevents the "experienced vi user" from becoming annoyed with the promptings. However, from initial observations and feedback, it is clear that even "experienced vi users" have learned something new from the system promptings, and it seems that most users benefit substantially from this scheme.

The knowledge that the system maintains on the user's capability is currently represented in a two level hierarchical network that is fixed and very specific to vi. At the highest level there is information on general capabilities, such as searches, insertions, deletions, the use of the assistance system, etc. At the lower level, information of specific capabilities is maintained.

The data on user's capabilities is maintained by using a short integer (0 to 255 range) for each node in this network. When a user issues a particular command the corresponding node in the network is incremented as the command is processed. This may have a ripple effect causing the values of nodes higher in the network to be incremented. This incrementing is not allowed to roll over, i.e. when the count reaches 255 it is not further incremented. The maintenance of the knowledge of capabilities is based on a simple temporal model that assumes that the user can forget capabilities of the system. This is achieved by periodically (once every two months) shifting all of the values in the user capability network one bit to the right. In this way if a user uses a command just once, and does not use it again before the periodic shifting, then the system would assume that the user had forgotten that command.

When the user initiates a vi session the system accesses the ".viknowledge" file and decides whether it should prompt the user, and if so, it decides what it should prompt him on. The prompting sequence is not strictly linear. If there is a negative response to a prompt the user is not prompted for the same item the next time, but that item is skipped and returned to later. The prompting sequence itself is dynamic and is based on an average over all current users. A command is very near the front if a high percentage of the users have used it; and commands with a lower percentage of use would occur later in the sequence.

The two major interactive systems under UNIX are the shell and vi. While eventual development of the interface could lead to an active assistance system for the shell (and therefore for UNIX itself), we felt that this was an infeasible initial goal, and thus made vi our first

project. Implementing this choice, however, necessitated compromises in our eventual goal of an independent active assistance interface. Since vi does not readily accept input from a pipe the overall design of Figure 1 required modification. The command processor, the routines for maintaining user capabilities and session characteristics, and the prompting mechanism were all integrated into vi. Only the Screen Display System was separated off as a separate process.

3. Screen Display System

The language processing portion of the system would ideally be a good natural language interface. However, these are difficult to construct and this was not our primary interest. We are initially satisfied with a menu based information display system that we call the Screen Display System (SDS). The basic design principles of SDS were borrowed from the ZOG system developed at Carnegie Mellon University [24]. The acceptance of this type of interface is clearly a compromise between the ideal system and a system that we could get functioning in a reasonable amount of time. For the generation of responses SDS uses a hierarchical network of menus or screens. These screens, following the design of ZOG, have a short text message followed by a set of possible next screens that contain further related information. Selection of the next screen to be displayed is performed by typing a single selection key. This, along with having each screen's message be short and easily read, allows the user to quickly move through the network and find the desired information. In addition to the next screen selectors that are displayed on the screen, there are a collection of standard selectors that are always available. These include:

(k) which accepts a collection of keywords and builds a list of screens, that have relevance to the set of keywords,

(d) which gives a more detailed explanation of the concept of the current screen,

(a) which gives a less detailed explanation (an abstract) of the concept of the current screen,

(b) which backs up to the previous screen (There is a stack of screens and this can be repeated until the stack is empty.), and

(n) which goes to a particular screen by name or by the name of its command.

4. Conclusions

In the past learning assistance or help systems have been passive in nature, and have not taken into account user capabilities. We propose an active assistance module that can be used as an interface to any software system. The active assistant has a component that "learns" user capabilities and session characteristics from his interactions with the system. This enables the interface to prompt a user suggesting more efficient and better ways to perform certain tasks. Unlike little used interactive tutorials such as the UNIX learn(1), this system will have a greater impact in educating users because it is integrated with the user interface. The system as a whole will provide the user with easy access to the knowledge needed to become an expert with the subsystems in the area that the user works, and save a lot of frustration when the user is having trouble in accomplishing a task.

5. Acknowledgements

The authors gratefully acknowledge the support of the NCR Corporation, Engineering and Manufacturing Division, Columbia, SC (ncrcae). We would also like to acknowledge and express our appreciation for the collaboration of Dr. Gautam Biswas on extensions of this work.

BIBLIOGRAPHY

[1] Beech, D., **Command Language Directions**, *Proc. of IFIP Workshop on Command Languages*, North-Holland, 1980.

[2] Biswas, G. and M. Matthews, "ORACLE: A Knowledgeable User Interface," submitted to COMPSAC 85.

[3] Boise, S. J., "User Behavior on an Interactive computer system," *IBM System Journal*, vol. 13, no. 1, pp. 2-18, 1974.

[4] Clancey, W.J., *Transfer of Rule-Based Expertise through a Tutorial Dialog*, Ph.D. Thesis, Stanford Univ., 1979. (also Tech. Report STAN-CS-769).

[5] Fikes, R. E., "Odessey: A knowledge-based assistant," *Artificial Intelligence*, vol. 16, no. 4, pp. 331-361,, North Holland, 1981.

[6] Finin, T.W., "Providing Help and Advice in Task Oriented Systems," *Proceedings of IJCAI-83*, pp. 176-178, 1983.

[7] Genesereth, M. R., "An Automated Consultant for MACSYMA," *Proc. Fifth IJCAI*, p. 789, 1977.

[8] Genesereth, M. R., "An Automated User Consultant for MACSYMA," Ph.D. Thesis, Dept. Computer Science, Harvard University, 1978.

[9] Griesmer, J. H., S. J. Hong, M. Karnaugh, J. K. Kastner, M. I. Schor, R. L. Ennis, D. A. Klein, K. R. Milliken and H. M. VanWoerkom, "YES/MVS:A Continuous Real Time Expert System," *Proceedings of AAAI*, pp. 130-136, William Kaufmann Inc., 1984.

[10] Hayes-Roth, F., D.A. Waterman, and D.B. Lenat, "An Overview of Expert Systems," in **Building Expert Systems**, pp. 3-29, Addison Wesley, 1983.

[11] Hayes-Roth, F., "The Knowledge-Based Expert System: A Tutorial", *Computer*, vol. 17, no. 9, pp. 11-28, 1984.

[12] Jackson, P. and P. Lefrere, "On the Application of Rule-Based Techniques to the Design of Advice-Giving Systems", *Intl. J. of Man-Machine Studies*, vol. 20., pp. 63-86, 1984.

[13] Joy, W., "An Introduction to Screen Editing with Vi," Unix Documentation, Univ. of California, Berkeley, 1979.

[14] Karna, K.N., "Artificial Intelligence and Man-Machine Interface", *Computer*, vol. 17, no. 9, pp. 8-9, 1984.

[15] Mark, W., "Representation and Inference in the Consul System," *Proc. Seventh IJCAI*, 1981.

[16] Mark, W., "Natural-language help in the Consul system," *Proc. AFIPS*, vol. 52, pp. 475-479, 1982.

[17] Matthews, M. and G. Biswas, "An Active Intelligent Assistant for Software Systems," submitted to the Seventh Annual Conf. of the Cognitive Science Society.

[18] Matthews, M. M. and Y. H. Kamath, "The FP-Shell," *Proc. 1984 USENIX Assoc. Meeting*, pp. 134-139, 1984. (Also submitted to Software Practices and Experiences.)

[19] Rayner, D., "Designing User interfaces for friendliness," *Proc. of IFIP Workshop on Command Languages*, pp. 233-241, North-Holland, 1980.

[20] Relles, N. and L. Price, "A user interface for on-line assistance," *Proc. Fifth International Conference on Software Engineering*, pp. 400-408, IEEE, 1981.

[21] Rich, E. A., "Building and Exploiting User Models," Ph.D. Thesis, Dept. Computer Science, Carnegie-Mellon University, April 1979.

[22] Rich, E. A., "Programs as Data for their help systems," *Proc. AFIPS*, vol. 52, pp. 481-485, 1982.

[23] Rich, E. A., "Natural Language Interfaces", *Computer*, vol. 17, no. 9, pp. 39-50, 1984.

[24] Robertson, G., D. McCraken and A. Newell, "The ZOG Approach to Man-Machine Communication," Tech. Report CS-TR-79-148, Dept. of Computer Science, Carnegie-Mellon University, Oct. 1979.

[25] Shrager, J. and T. Finin, "An expert system that volunteers advice," *Proc. AAAI*, pp. 339-341, William Kaufmann Inc., 1982.

[26] Sleeman, D. H. and M. J. Smith, "Modeling student's problem solving," *Artificial Intelligence*, vol. 16, pp. 171-188, North-Holland, 1981.

[27] Smith, D. C., C. Irby, and R. Kimball, "The star user interface: an overview," *Proc. AFIPS*, vol. 52, pp. 515-528, 1982.

[28] Stabile, Lawrence, "Frame Based Computer Network Monitoring," *Proc. AAAI*, pp. 327-329, William Kaufmann Inc., 1982.

[29] Temin, A. and E.A. Rich, "Mirror: A Language for Representing Programs for Reasoning," *John Hopkins Univ. Conf. on the Role of Knowledge Representation in Problem Solving*, Baltimore, 1984.

[30] Temin, A. and E.A. Rich, Representing Programs for Reasoning, Department Computer Science University of Texas at Austin, 1985.

[31] Tou, F. N. and M. D. Williams, "RABBIT: An intelligent database assistant," *Proc. AAAI*, p. 314,, William Kaufmann Inc., 1982.

[32] Wilensky, R., "Talking to Unix in English: An Overview of UC," *Proceedings of AAAI-82*, Pittsburgh, pp. 103-106, 1982.

[33] Wilensky, R., Y. Arens, and D. Chin, "Talking to Unix in English: An Overview of UC," *Communications of the ACM*, vol. 27, pp. 574-593, 1984.

UNIX† Tools for a Personal Database

Michael J. Hawley

Computer Division
Lucasfilm, Ltd.
P.O. Box 2009
San Rafael, CA 94912

ABSTRACT

Progress in database and interface technology has made interesting new information retrieval tools feasible. We have combined a simple set of UNIX database commands for storage and keyword-retrieval of files with a fast interface for browsing and editing text. This permits both keyword retrieval and menu exploration in the UNIX directory space.

The database programs maintain a hashed, inverted index for content-addressable file retrieval. Files can be referenced using boolean combinations of keywords instead of shell-style regular expressions. This is surprisingly useful in practice:

```
p scream crunch/nose
```

means "print out all files containing `scream`, and all files containing `crunch` and `nose`"

The text-handling interface exploits a *Frame* abstraction. This affords fast perusal and editing of text on a bitmapped display, with multiple windows, high-level views, scrolling, cut/paste editing, regular expression searching, etc. Directory hierarchies can be graphically represented. Besides browsing database and file systems, such an interface is amenable to other common information retrieval tasks, like reading mail or news. Adding an extra "button" for sound playback, we use it for exploring a large, digitized sound effects library.

This paper describes the design and use of these tools.

1. Introduction

At Lucasfilm we have a variety of "database" problems, including management of personal and office files, browsing library catalogs, and keeping track of film production data. Retrieval is usually the critical task. For instance, users often need quick answers to "random fact" questions like "get me directions to Fantasy studios," or "what's the phone number of *Cellar in the Sky*," where the required information is found somewhere in a personal or public database. Or, a sound designer might want to find and audition some nose crunches and *wurfs** in a large sound effects library, by typing a few keywords and touching some graphical buttons on a touchscreen.

For such retrieval tasks a *grep*-based application is usually too slow – we need an index of

† UNIX is a trademark of AT&T Bell Laboratories.

* A *wurf* is a stomach punch noise, in film sound jargon.

some kind. Furthermore, when data is well-structured (like our sound effects library, which is a bushy UNIX directory tree) users will – optionally – want to browse the tree branches. We have addressed these issues by combining some programs that implement content-addressible (*i.e.*, keyword) retrieval of files with a fast text browsing and editing interface. The design of the database programs is motivated by cognitive factors in information retrieval, as well as the desire for small, fast tools that mesh cleanly with the UNIX environment. Landauer *et al.* have shown extensively how imprecision in the way humans name things severely limits the chances for successful keyword retrieval, and how rich indexing techniques provide the best known method for improving a user's chances [3,4]. Remde [10] has done a first implementation of a content-addressable filing system which has proven very useful. A few others [1,2] have discussed related methods, but do not fulfill our needs. Lesk and Geller [5] have analyzed tradeoffs in menu and keyword retrieval paradigms. Following such leads, we have built our tools.

2. Database Programs

The database programs maintain an inverted index that associates *keywords* with lists of *objects* to which they refer:

keyword → *object1 object2 object3 ...*

To add a *keyword*→*object* association, *object* is simply prepended to the *keyword*'s object list. This is a reasonable default: objects are retrieved in "natural" LIFO order, and it is the oldest associations that will be "forgotten" if the list overflows.† Objects contain a file name (to which *keyword* refers), and various other data, like write date, status bits, etc.

The index is extendibly hashed using a modified version of **dbm**(3), which provides fast hashed-key access to a potentially enormous number of associative key/content pairs. We are also experimenting with a new hash algorithm devised by Ryan [11], which appears to be significantly faster than **dbm** (roughly 2-3x faster for writes, and 20x faster for retrievals and traversal in this application). Other low-level storage schemes can certainly be used; hashed files are minimal, general, and fast. Boolean keyword combinations are computed by merging object lists (taking unions and intersections as appropriate), and objects are represented internally as arrays of numbers, so comparison is quick. Our library allows several hashed files to be opened simultaneously. This permits path-like searches through lists of indexes. Because all objects in the index are ultimately *files* (as opposed to some structure completely internal to the database), the programs combine well with other UNIX facilities. Issues of file protection in the database are subsumed by the operating system, for instance, and cleaning out a database or rebuilding it from scratch is easily done with a two or three line shell script. This outweighs the expense in *i-nodes*.

2.1. Hm... uh... p... grok.

For personal use – to help indiviuals store and retrieve loose notes and jottings – there are two basic programs, `hm` and `p`. `Hm` takes input (from files, or *stdin*), squirrels it away somewhere if necessary (*e.g.*, as a uniquely-named file in `$HOME/.db`), extracts all the keywords from it, and adds any new *keyword/file* associations to the user's index. The default keyword extraction routine is rather uninteresting. It does case folding and suffix stripping and uses a *stop list* of common, non-key words; command-line options provide some flexibility. This results in about 3 bytes of index per byte of stored data. To "remember" something, a user types:

† In our current implementation, a maximum of about 500 references per keyword is permitted. We rely on richness of the index and the ability of users to choose reasonably precise keys (which they almost always do). For modest databases, keywords that require so many associations are hardly precise, and probably warrant some kind of special treatment anyway.

```
$ hm
remember to phone Max (583-3945)
or Lloyd about force-sensing touch screen
^D
Hm..... okay.
```

And to retrieve it (print it on the standard output):

```
$ p phone
```

Of course, this retrieves *all* files mentioning `phone` (with the reminder about phoning Max appearing first, if it was most recently entered). To be more precise, one could request `p force/touch/screen`, where "/" connotes boolean *and*. On the other hand, the command `p force touch screen` prints all files mentioning `force`, then all that mention `touch`, then all that mention `screen`. `P` also accepts a few options to print just the matched filenames, or short descriptions, instead of the full text. Day to day, this simple solution is surprisingly handy.

There are other low-level tools useful for building applications. For instance, the command

```
assoc [-i index] key [key...] file
```

adds new *key→file* associations to an index; similarly, `unassoc` breaks them.

The command

```
getkeys [files...]
```

extracts keywords from its input and prints them on the standard output. There is also a `-ldb` library of `C` routines. The `p` command, which uses this library, is about 80 lines of `C`, and not much bigger in binary size than some versions of *cat*. Personal database files live in a directory called `$HOME/.db` by default. They can still be searched or printed by normal means; *e.g.*, `cat .db/*` dumps all the files in a user's database to the standard output.

Given such tools, it is easy to tailor other applications. A small browsing program, `uh`, provides a line-oriented command interface and is useful for groping around in a personal database. A one-line shell script, called `remember`, simply funnels its input off to a `hm` process which finishes in the background on the user's primary host, so that text can be saved conveniently from remote machines. The notion of "search path" (a list of hashed indexes which are searched during retrieval), makes it easy to explore other pools of knowledge. For instance, there is an all-purpose command for getting in touch with information:

```
grok [things...]
```

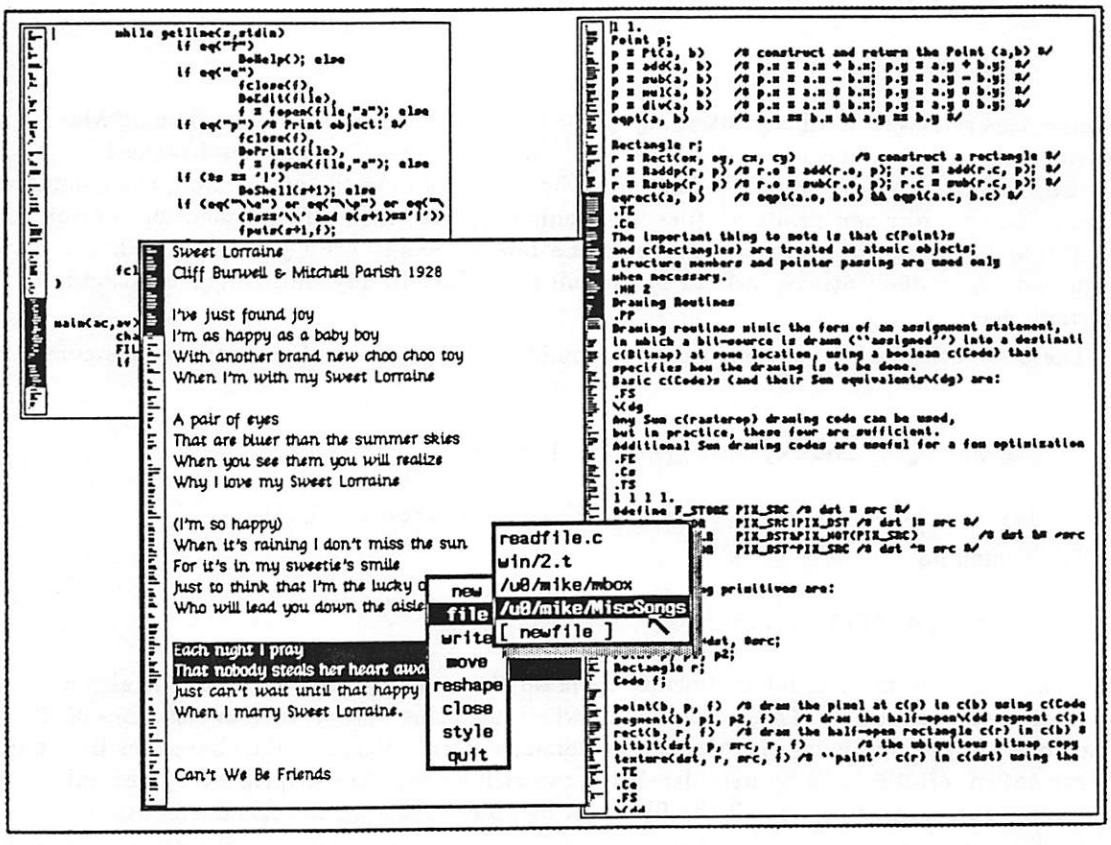
`Grok`† is a short shell script does one of several things. If `things` are user names or directories containing indexes, `grok` simply adds the appropriate indexes to the database search path — a way of "picking" other users' brains. If `things` are unindexed files or directories, it will assimilate them into the user's personal index (after a query). If `things` are unknown (or omitted), `grok` simply executes `p`, responding to user keyword queries.

---

† "observer interacts with observed through the process of observation. *Grok* means to understand so thorougly that the observer becomes a part of the observed — to merge, blend, intermarry, lose identity in group experience" — Robert Heinlein, *Stranger In A Strange Land.*

3. Frames

Frames are a popular and powerful abstraction for dealing with text, derived from Macintosh, Smalltalk, and Blit systems [6,8]. They provide text selection and cut-and-paste style editing using a mouse or touchscreen, scroll bars, multiple windows for working with several files, various display styles (multiple fonts, wrapped or truncated lines, "bird's eye views"), regular expression searching, and so on. A frame is a rectangular window which displays part of a text buffer (*i.e.*, file). A frame editor looks like this:



Selected text is shown in reverse video (black) and can be edited, written to files, filtered through pipes, etc. The black box in a scroll bar indicates the size and relative position of the frame on the file. The scroll bars may optionally contain "bird's eye" views of the text, which often reveal interesting and useful patterns (in the example above, C functions, song verse shapes, and paragraph structures can be clearly seen). Scrolling uses a mouse button paradigm similar to Smalltalk and Blit versions. With the mouse in the scrollbar, the middle button lets the user move to any point in the file by sliding the black box (the choice is made on release of the button). The right button scrolls down (forward in the file), and the left button scrolls up. The amount scrolled is determined by the distance from the mouse to the to the top of the frame: clicking near the top scrolls a few lines, clicking in the middle scrolls half a page, clicking at the bottom scrolls a whole page.* When scrolling, the left and right mouse buttons repeat while held, so sliding the mouse up and down effectively changes the rate (and the direction, if you move it far enough). Popup menus contain editing and frame management commands; these can also be bound to the keyboard using macros. By choosing appropriate font and scrollbar sizes, the essential parts of this interface work adequately in a touchscreen environment, too (thought not as well as a mouse).

* Or, looking at it another way, the right button moves the line opposite the mouse to the top of the frame, and the left button inverts this.

Performance (on a Sun under the Lucasfilm window system[7]) is surprisingly quick, considering the relatively naive implementation. A `Frame` is just a doubly linked list of lines:

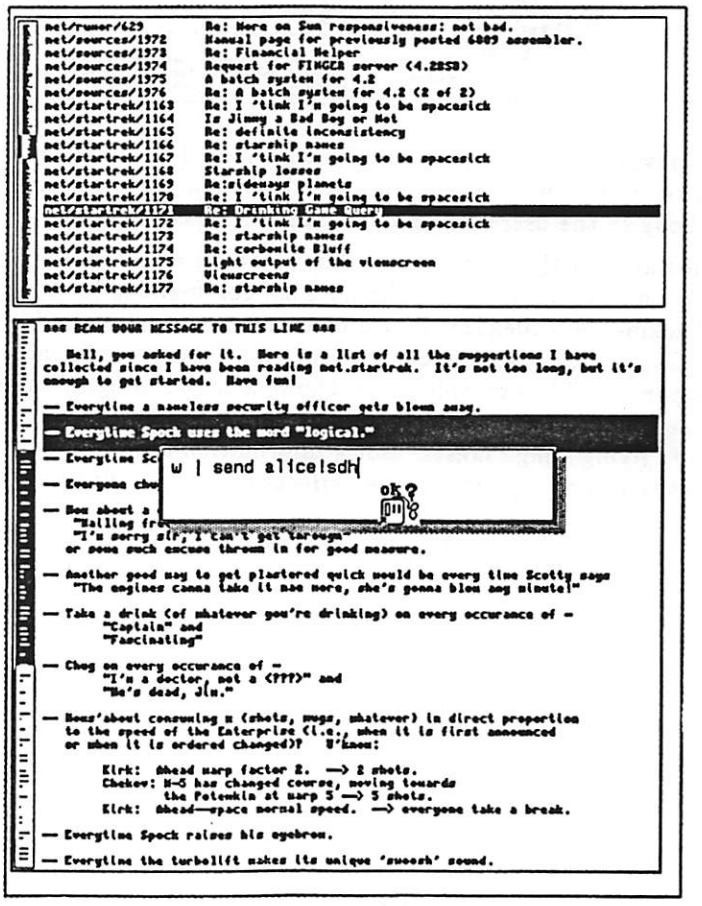
```
typedef struct Line { char *s; struct Line *prev, *next; } Line;
typedef struct { Line *l; char *s; } TextPt; /* location (point) in a buffer */
typedef struct { TextPt a,b; } Selection;    /* two pts are a selection */

typedef struct {        /* the text frame representation of a file */
    Line *l;            /* buffer */
    int n;              /* number of lines in l */
    Line *topline;      /* current top line in frame */
    int toplinen;       /* number of top line */
    Selection s;        /* current selection */
    Font *f;            /* font to use for this frame */
    Bitmap *b;          /* saved image of frame */
    Rectangle r,        /* rectangle on screen */
              sr,       /* scroll rectangle */
              sbr,      /* black scroll box */
              tr;       /* rectangle containing text */
    char *name;         /* filename (if any) */
    ...
} Frame;
```

Routines for manipulating frames are straightforward. Inserting and deleting characters typed at the keyboard is done using the general `Cut()` and `Paste()` routines, for instance. It is obvious that optimizations can be made, but interesting that a direct approach works so well.

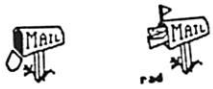
3.1. Frame Applications

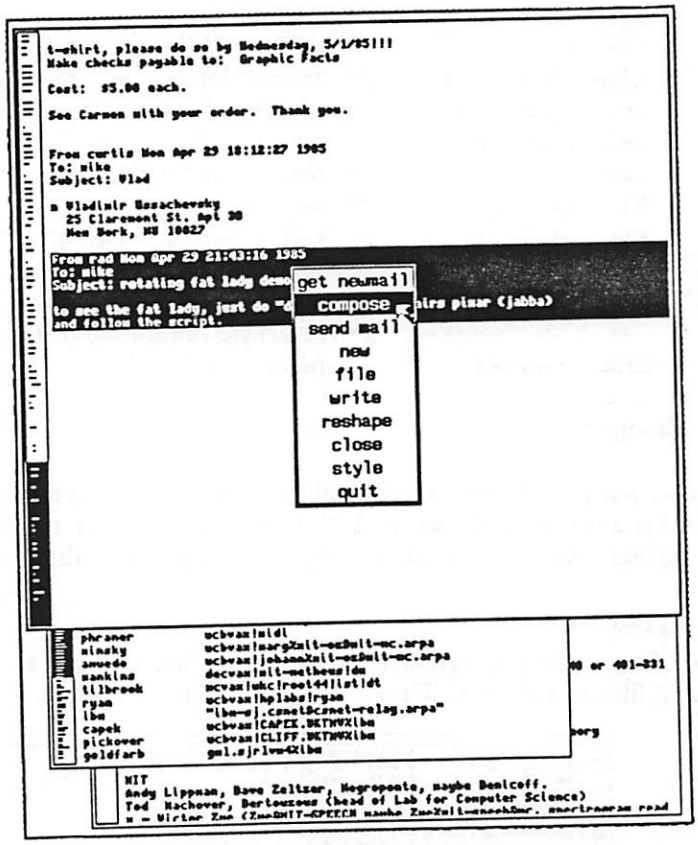
The basic frame editor interface is fairly easy to use in other applications because it is available as a `C` library resource. Here is a USENET news browser:



To browse the news, a user types `news`. This is a 5-line shell script that executes a variant of *readnews*(1) on a large host, creating a file of headlines on the local workstation. If there is news, a frame-based program called `browsenews` pops up. The user sees a little newspaper icon which expands with a click to reveal the headlines. Reading the news is just like editing text, except selecting a headline (by holding and sweeping, or double-clicking) causes the article to pop up in a browsing frame: `Browsenews` requires 40 lines of `C` plus the frame library.

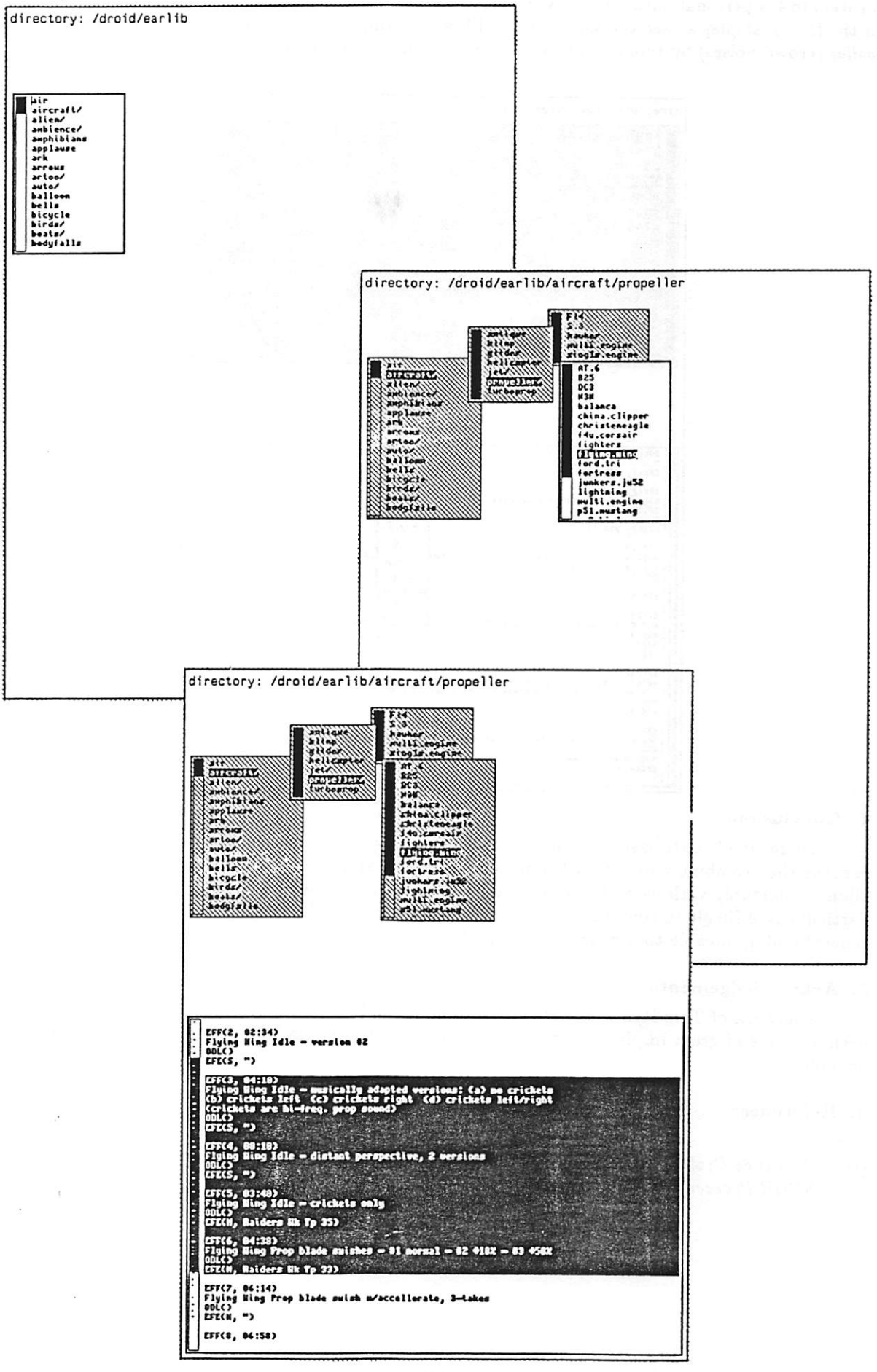
`Mail` is another instance of the frame editor, with the addition of three menu commands — `get new mail`, `compose`, and `send` — and a little mailbox icon with a flag that flips up when new mail arrives:



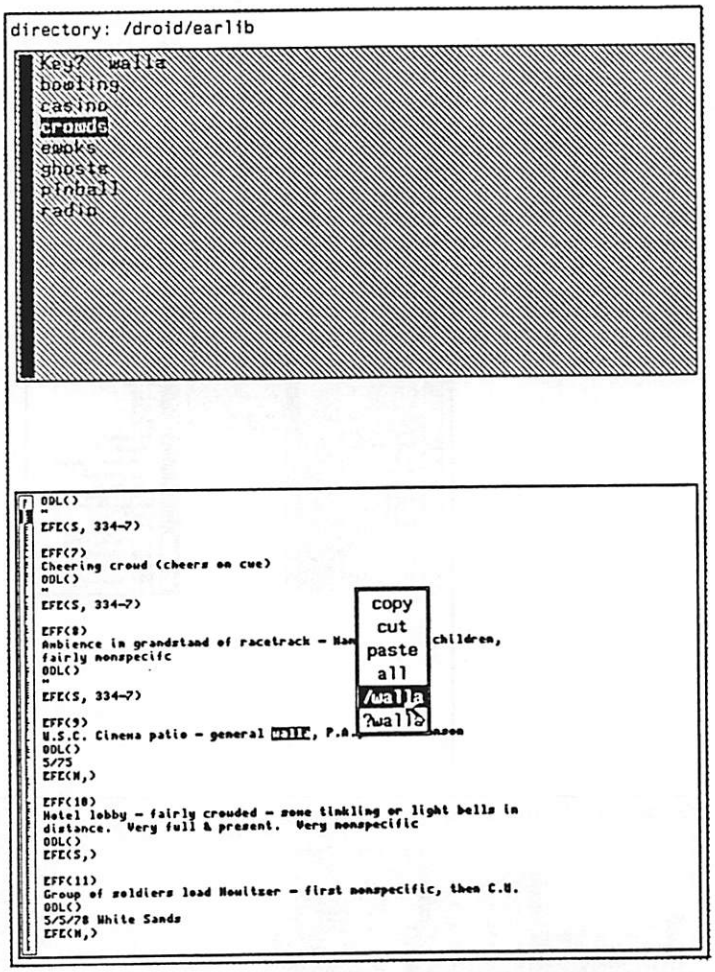


Each of these commands invokes a user-settable shell command. For example, the `compose` command (to compose a reply or new letter) filters text through a program that reverses the header and indents the body so the user can edit a message with context.

`Dir`, yet another instance of the frame editor, knows about directories. When a user attempts to open a frame on a directory, an arbitrary shell command is invoked to obtain a listing (*e.g.*, `ls -F`). Clicking on a filename in a directory frame causes the file to pop up in a browsing frame, below. Clicking on a subdirectory name causes the new listing to pop up in a frame to the right of its parent. The example below shows a sound designer browsing the tree in search of some airplane sound effects. After browsing through a few obvious directories, the designer selects the file of "flying wing" noises. Since information about how to play sounds is embedded in the text it is trivial to pipe the desired effects to a filter that plays them.


directory: /droid/earlib
air
aircraft/
alien/
ambience/
amphibians
applause
ark
arrows
artoo/
auto/
balloon
bells
bicycle
birds/
boats/
bodyfalls
directory: /droid/earlib/aircraft/propeller
antique
blimp
glider
helicopter
jet/
propellers
turboprop
F14
S.3
hawker
multi.engine
ztoglz.engine
AT.6
B25
DC3
H3H
balanca
china.clipper
christeneagle
f4u.corsair
fighters
flying wing
ford.tri
fortress
junkers.ju52
lightning
multi.engine
p51.mustang
directory: /droid/earlib/aircraft/propeller
EFF(2, 02:34)
Flying Wing Idle - version 02
ODL()
EFE(5, '')
EFF(3, 04:10)
Flying Wing Idle - musically adapted versions: (a) no crickets
(b) crickets left (c) crickets right (d) crickets left/right
(crickets are hi-freq. prop sound)
ODL()
EFE(5, '')
EFF(4, 00:10)
Flying Wing Idle - distant perspective, 2 versions
ODL()
EFE(5, '')
EFF(5, 03:40)
Flying Wing Idle - crickets only
ODL()
EFE(N, Raiders Mk Tp 35)
EFF(6, 04:38)
Flying Wing Prop blade swishes - 01 normal - 02 +10% - 03 +50%
ODL()
EFE(N, Raiders Mk Tp 37)
EFF(7, 06:14)
Flying Wing Prop blade swish w/accellerate, 3-takes
ODL()
EFE(N, '')
EFF(8, 06:58)

At any time, of course, text can be written to, or read from, a pipe. A user may file away an item in his personal database by writing some text into `remember`, or retrieve possibilities. In the last example, we see the sound effects library again, but this time the user is looking for *wallas* (crowd noises) by typing a keyword, then clicking on a matched file:



4. Conclusions

These simple database tools have been enormously useful in everyday work. This is partly because they combine well with other system resources, and partly (we believe) because they align comfortably with users' basic cognitive needs. None of the applications presented here are particularly difficult to construct, and the methods — extendible hashing and text frames — are general and applicable to a wide range of problems.

5. Acknowledgements

The work of Tom Ryan, Rob Pike, Joel Remde, and Tom Landauer, among others, has been a source of great inspiration, and the resources at Lucasfilm have made much of this work possible.

6. References

[1] W. Bruce Croft, "Applications for Information Retrieval Techniques in the Office," *ACM SIGIR Proceedings* (Summer 1983) p18.

[2] R. G. G. Cattell, "An Entity-Based Database Interface," Xerox PARC CSL-79-9 (August 1979).

[3] Susan T. Dumais, Thomas K. Landauer, "Using Examples to Describe Categories," *ACM CHI '83 Proceedings* (December 1983) p112.

[4] G. W. Furnas, T. K. Landauer, L. M. Gomez, S. T. Dumais, "Statistical Semantics: Analysis of the Potential Performance of Key-Word Information Systems," *Bell System Technical Journal* **62**(6) (July-August 1983).

[5] V. J. Geller, M. E. Lesk, "User Interfaces to Information Systems: Choices vs. Commands," SIGIR Proceedings (Summer 1983) p130.

[6] Adele Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley (1984); esp. Ch 3.

[7] Michael Hawley, Samuel Leffler, "Windows for UNIX at Lucasfilm," *Usenix Summer Conference Proceedings* (1985).

[8] Rob Pike, "A Text-Oriented Terminal Multiplexor for Blits," *Usenix Summer Conference Proceedings* (1984) p173.

[9] Rob Pike, "The Blit: A Multiplexed Graphics Terminal," *AT&T Bell Laboratories Technical Journal* **63**(8) (October 1984).

[10] J. R. Remde, "PDB - A Content Addressable Personal Filing System," Bell Laboratories Technical Memorandum (October 1983).

[11] T. Ryan, unpublished work.

Thoughts on an All-Natural User Interface

Marion O. Harris

Bell Communcations Research
445 South St., Morristown, NJ 07960
(201) 829-4309
bellcore!moh

ABSTRACT

The language of the command interface is technically similar to English, and probably should be more so. This paper discusses the similarities, and urges that they be consciously recognized and encouraged as our interface develops. At the same time, it explains why the author does not preach the complete "natural language" interface. There are some clues to the process of naming commands and options, and a few strong words on the subject of new users.

How should we talk to computers? Some say in "natural language", others in short forms designed for efficiency. Most agree that we should change current practice; I do not. I shall try to show that we already speak a form of natural language to computers, and can benefit from recognizing the fact. Then we can take advantage of human linguistic habits to guide the design of interfaces.

The first decision a designer must make is whether to accept free-form conversational English (or whatever). Probably no computer *system* does this. Individual programs do it - or their proponents say they do - but the ones that claim high success rates are all severely limited: they deal with single databases, or they answer queries on very small universes. Furthermore, they do not toss a conversational ball back and forth with the user, who is usually restricted, after the opening line, to one-word or even one-character replies to canned questions or menus. In all cases the failures are not like erring humors - oops, sorry, human errors, and the machine's inflexibility requires users to write so formally, so accurately, as to lose all sense of ordinariness. Still, some say that our system interfaces should accept whatever is typed at them. If you are such a philosopher, this paper is not for you.

Well then, assume that we will ask people to meet their computers half-way. What resources do they bring to the alliance? People have vast experience with symbol systems: they use telephones and telephone books, with area codes; play bridge; address letters, with zip codes; pay bills, write checks, file tax returns; read music; follow maps; speak; read; write. Compared with these, learning the language of computer commands is duck soup.

The reason that commands are simple to learn is that they draw on language, our most nearly universal symbol system. To the extent that they are difficult, it is because they depart from language in unexpected ways which trip us up.

Natural language, like other codes, has two parts: a set of individual symbols, and the rules for their use. The symbols are, more or less, words; the rules are the syntax. In the background is the world of facts that provides language with a context. When two people talk, they understand each other through a common vocabulary and syntax, and also because their contextual worlds overlap. Beyond that, each partner can bend to the task of understanding, asking questions to clear up ambiguity, allowing for mistakes or variations in performance, watching gesture s and facial expressions. Conversational language, for its part, has grown in response to human flexibility. It is redundant on every level: speech sounds contain more features than the phonemic system uses, words are more than minimally distinct, inflectional endings occur in agreement-sets, syntactic cases are marked by both association and word order. Yet this same device, stripped of its redundancies, is the language of computer commands.

All-Natural Syntax

How do we strip down natural language to arrive at a command line? First, we toss out the syntax, planning to arrange the symbols in some different pattern that the computer expects. We describe this pattern in mathematical terms: the command name is the function, the other symbols are arguments "to" it. But we can also describe the pattern in traditional grammatical terms. Consider

sort -n -o splat messyfile

We could assign these grammatical functions:

sort	**-n**	**-o splat**	**messyfile**
¦	¦	¦	¦
verb	**modifier (adverb of manner)**	**indirect object**	**direct object**

In English, we would say

Sort messyfile onto splat in numerical order.

The chores performed by syntax are identical in the two cases, and so are the devices used to perform them, word order and marking.

Each computer command is the name of an act that we want performed, and convention dictates that this name appear first on the command line. Similarly, English imperative sentences begin with the verb, which names the request. Syntax for the rest of the "sentence" conforms to an assortment of customs and contexts: according to the environment, you might or might not choose to say

sort -n messyfile -o splat

or

sort messyfile numerically onto splat.

or

sort messyfile -n >splat

or

sort my file called messyfile on a numerical basis,
writing the results on my file called splat.

(Of course, you could start with an adverb - and of course, you could install a command-line parser that would accept arguments and command names in any order. We have exchanged this bit of freedom for a bit of speed.) Every language makes some use of word order, if only to pay it no heed; most users come prepared to understand it.

The second syntactic device, marking, is equally familiar, though it requires learning what the markers are. In Unix computer command syntax we use +, -, =, >, or < (and other symbols as well) to flag the purposes of 'words' on the command line. In English, **-ly** is the sign of an adverb, **to** marks the indirect object, other prepositions label other grammatical cases, and so on. The absence of a mark can have meaning as well. In English, a noun is often marked only as to

its part of speech, by an accompanying article, or by a plural or possessive suffix. In that case, word order determines its function as subject or object in the sentence. In computer command language, we may have the convention that an unmarked argument is an input file; or else, if word order is made to matter, or if no file name could compete, the argument may be some other important token, like the search string for the Unix™ pattern-matching command **grep** or the addressee of **mail**.

The abandoned syntax of natural language, then, appears to have turned up again on the doorstep. The design uses word order, more strictly than does English, and marking, with a far more limited set of markers. Why is this very simple syntax enough? Because the job is so simple. Yes, we have banished poetry, but we have also banished time, uncertainty, opinion, hope, and assertion of fact. The parser need never handle "If I were you, and I think I am ..." As a final touch, the "subject" of a computer command is never in doubt, the divine "understood" whose name cannot be invoked. If we find ourselves also doing without syntactic gingerbread like articles and plurals and gerunds, it is only because in this primitive world there is no work for them.

All-Natural Names

The other component of a language is its vocabulary of symbols, names of things it refers to. The naming process organizes a piece of experience. It picks out some set of facts and spotlights them in the foreground for our attention. Consider, for instance, the words *rush* and *dawdle*. Both imply movement, and apparently both assume that we have in mind a concept of normal speed. *rush* points to speeds higher than normal, *dawdle* to lower. In the background may lurk, for all we know, debt, villainy, or terminal cancer, but for the moment they are not onstage. Sometimes, when intuition tells us that a name is "wrong", we realize that it spotlights unsuitable pieces of experience. In literature, the result may be poetry, but science must resist.

The designers of the first ancestral set of commands were already influenced by a teeming linguistic tradition. By now, the jargon of computing trails its own clouds of association. Natural languages and technical languages grow in different ways, but they both do grow. Each is a system, with the meaning of every token held in tension by the meanings of all the rest. Wherever a new token is introduced, the tensions shift. In the natural world, the process is continuous and largely unconscious. Sometimes a new word is introduced, consciously or even with fanfare, like *cybernetics*, or, like *Unix*, with regulations designed to protect its money-making potential. More often, though, people just start talking, and linguistic boundaries begin to blur. Technical languages are more subject to regulation. For one thing, their terms are introduced by the field's practitioners, immediately after a phenomenon appears or sometimes even beforehand. For another, they are chosen so as not to overlap, not to be vague, not to have multiple references. People use them carefully, thus restraining a good deal of their drift around the semantic landscape. Programming language cannot be said to drift at all; instead, it requires years of conference or of petition to the designer before a change happens. Command languages, though controlled, are not so resistant to change, because they have one foot in the natural world.

The interface designer can take advantage of this position in no-man's-land. On one hand, users' associations with the vocabulary of natural language will transfer to the command language if they can, thus giving it a comfortable home in memory. On the other hand, when a new function appears, its name can be established overnight, by fiat.

Take the example of *grep*, a nonsense word that has become famous, first locally as the name of a Unix command, and then through the public relations efforts of Donald Norman of the University of California, San Diego (*Datamation*, November 1981). The word originally sprang full-blown from the head of Ken Thompson, who had just written a regular-expression searcher and needed a name. Later it was "explained" as being a sort of acronym for the **ed** instruction **g/regular expression/p** (which prints every line containing the expression). Norman says, "What

does grep mean, you may ask? 'Global REgular expression, Print' - at least that's the best we can invent." Not bad, for a guess in the dark. He goes on, "The name wouldn't matter if grep were something obscure, hardly ever used, but in fact it is one of the more powerful, frequently used string processing commands [on Unix]." But what we *may* ask is, how does it matter? Anyone who uses it often must know it. A new user who wants to search for patterns looks up those words in the system manual's permuted index, or asks a friend, and then, whatever the command name turns out to be, starts to remember it. What is the memory process like? We have some clues from the things people say, the ways that computer vocabulary turns up in conversation. The Unix commands **cat, grep,** and **ls** are famous for their arbitrary names, but users have no trouble living with them. "I couldn't even do an ls on that file," they say, "Just cat it out on your terminal; try grepping through your mind ..." Syntactically, computer commands drop into the verb slot of natural language, taking on the inflectional suffixes, class cleavage patterns, and agreement conventions of ordinary speech. Semantic associations begin to vibrate, too, as the jokes take over: "In my head it's more like groping; don't let that cat out on my terminal".*

The point is that the new name of a new thing is easy to accept. After all, suddenly one day there was Kleenex. Problems arise, though, when old names are redefined for technical purposes, or when familiar functions are renamed. Usually the change is only partial, so that the mossy old associations trail along but find no rocks to gather on. This is called 'proactive interference'. Better not muddy the bathwater.

Designing for Naturalness

A comfortable interface must sit on a rationally designed system. We have learned from Unix the pleasure of working on a clean system, where each command performs a single, low-level activity. Then users can assemble their own mix-and-match tool boxes, without having to dodge around other people's decisions about which options are good for them. The designer who starts from scratch has an ideal chance to carve up the world of machine activities and assign distinctively-named commands. The naming process actually helps to find natural boundaries between activities; often the name of a command comes to mind before the command is designed. If a command is difficult to name, that can be a danger sign of improper design.

A crucial lesson from Unix: there should be only one way of doing each chore, one command per activity. (The hard job, often, is to isolate single activities.) By contrast, the same option-name, like an adverb, can be used on many commands, a mnemonic practice. When the question is whether a particular activity is a command or an option, it may help to test it as a possible 'adverb of manner', like 'without sending me mail' or 'deleting the original in the process'. If the parallel applies, the activity is probably an option, a *way* of doing things rather than a deed in itself.

Command designers hear often from the keystroke-counters, who believe that a character saved is a user soothed. Their point of view is that option and command names should consist of the fewest possible characters, preferably one or two. 'Natural' thinking disagrees. Before users can begin to type, however fast, they must search their [human] memories for the command name. The search will work better if it can depend on ordinary paths for word recovery - semantic networks, if you like - rather than on the limited set of associations by which "pure" arbitrary symbols are fixed in the mind. It is probably true, particularly for constant users of the system, that the one or two commands that they call most frequently could be named anything at all.

---

* It turns out that there is technical justification for the name *grep*. A semi-formal linguistic unit called a *synestheme* is defined as a sequence of sounds, smaller than a morpheme (which is, roughly, a word), that is used semantically in some manner intuitively consistent though difficult to define. The sequence *sl* is the classic example, appearing in such words as *slip, slime, slither, sloppy, slum*, etc. In support of a *gr* synestheme we can adduce *grab, grapple, grasp, grip, grope, grub, graft*, and perhaps even *grit, grimace, grid*, and *graze*.

But those frequent commands will not be the same for every user, who still has to remember how to use the rest of the system.

One compromise is to permit the user to truncate the command name. This takes advantage of normal linguistic memory while saving the fingers. It requires that all commands on the system be unique within the truncation length, not a serious difficulty if the length is, say, four characters. On the other hand, few command names contain more than six or seven characters at most, and one suspects that the extra effort of remembering the full name and then amputating its tail may at least offset the saving in muscular energy. A practiced typist has real trouble interrupting a familiar pattern of strokes. But people who have never learned to type may be grateful.

A perennial problem in system design for more or less public access concerns the variety of experience among users. Screams of pain come primarily from two sources: people who aren't getting what they're used to, and people who don't know what to expect. Administrators should probably listen to the two kinds of complaint in different ways. Experienced users will make their own adjustments, though their complaints can be helpful.

The real problem is the beginning user: what can or should be done to adapt the system for him? I believe the answer is a resounding "nothing". It makes no sense to cripple sophisticated users by designing a system for neophytes, unless neophytes are the only users it will ever have, as in the case of publicly located information terminals. Instead of limiting the system, you can provide everything possible to help new users grow up. Certainly there should be clear, simple tutorials in the very fundamental procedures like calling up, logging in, creating files, using the basic commands - these must be available, too, for infrequent users, who are perennially nearly new but whose memories are easily jogged. In appropriate situations, courses may help. But the one most important aspect of training is to provide answers to the new user's questions the instant they are needed. To do that, there is no substitute for a human being. After all, how did you learn to phone? to drive? to talk? I would strongly recommend a mentor program, or the provision of a "buddy" by the user's own work area, as the investment of time and assets likely to bring the greatest reward. It might also be helpful to offer a kind of "learning shell", a playpen program that gradually lowers its bars and dangles new toys to stretch the beginner's horizons. But normal performance of the system itself, the way its commands are named and used, the amount of conversation it requires, should be geared to the standards of the most proficient users in the stable. They work in a way that others should be imitating.

Conclusion

To sum up the recommendations in this paper, I am urging the continued use of our present type of command language derived from, but not coextensive with, English. Once we recognize the ancestry of the naming principles we have been using, I believe that we can consciously depend on them to name things in the future. Then command usage will become increasingly natural, and so easier to remember.

I am urging those responsible for the user interface to take a system-wide view of the capabilities within their domain, and to divide them among commands in such fashion as to profit from functional and linguistic boundaries already in existence, on the system and elsewhere in human experience.

I am urging namers to remember that words and things come complete with associations. These can help if we do not try to violate or rearrange them. I am urging authorities to make special provision for new users, with an eye to increasing their proficiency as fast and as pleasantly as possible. The assistance should certainly include clear, informal documentation as well as special tools on line, but above all it should offer constant human support during the crucial early learning period.

A XINU Virtual Machine

Jonathan Bachrach
John Wallerius
Jehan-François Paris

Department of Electrical Engineering & Computer Sciences
University of California, San Diego
La Jolla, CA 92093

ABSTRACT

XINU is a simple multi-tasking operating system designed to run on a set of LSI-11's linked by a store-and-forward ring network. It was designed to provide an environment where students could learn operating system concepts by working with a well-documented, complete system. We present here a XINU virtual machine running on a VAX architecture under Berkeley 4.2 UNIX.† Our paper discusses the XINU port and the enhancements that were added to the original XINU system. These enhancements include an interactive shell, a debugger and enhanced networking facilities.

1. INTRODUCTION

Many operating system concepts are better understood if students can have hands-on experience with a complete running operating system. Students should be able to see how functions like process management, memory management, interprocess communication, and so forth are implemented in a real system. They should also be given the opportunity to experiment with the system by modifying some of its functions and evaluating the impact of these modifications on the system performance.

Traditional multi-tasking operating systems are not suited for this purpose. They are too complicated and often contain proprietary information. Besides, any attempt to modify the system kernel would result in having the machine unavailable for other uses for long periods of time, to say nothing of the security risks involved. These considerations have led to the emergence of a new type of operating systems that are specially designed to be teaching tools. These so-called "toy operating systems" are much smaller than regular systems. Since efficiency is not anymore a key consideration, the various functions of the system can be implemented in a straightforward way, so as not to confuse the novice.

The TOY Operating System developed by Fabry et al. at the University of California, Berkeley, is a prime example of such a system [Fabr83]. TOY runs as a user process on a host computer and simulates a fictitious machine with a very simple instruction set. Since the TOY operating system is itself written in a high-level language (C in the current implementation), students can quickly learn to modify it. The impact of these modifications can be later assessed by running user programs on the modified system. Since the TOY virtual machine is simulated at the instruction set level, all TOY machine instructions are interpreted by software. A very large range of memory management schemes, which includes virtual memory, can thus be simulated.

† UNIX is a Trademark of Bell Laboratories.

I am discouraging the idea of a "natural language" interface that lets users type anything that they might speak. Technology is very far from handling the job, and so much freedom is not necessary.

With these considerations in mind, I believe that we can design clear, pleasant command lines that need no artificial preservatives on the shelves of memory.

The TOY system lacks, on other hand, any compiler or assembler. As a result, all user programs need to be written in the TOY machine language, which might soon prove to be a rather tedious task.

The HOCA system developed by Babaoglu et al. at Cornell University [Baba83b] does not suffer from this limitation. HOCA runs on the CHIP virtual machine, which is a slightly modified PDP-11 [Baba83a]. CHIP and HOCA were written to run on a VAX 11/780. They can thus take advantage of the existence of a PDP-11 compatibility mode on that architecture.

Like TOY, HOCA lacks any facility for implementing distributed systems. We felt this to be an important limitation in an environment like ours where students typically take a two-course sequence in operating systems with the first course dedicated to traditional operating systems and the second one covering distributed systems. We thus decided to turn to a third educational system, XINU [Come84]. XINU was designed by Comer at Purdue University to run on a set of LSI-11's linked by a store-and-forward ring network. It is written almost entirely in C and is exceptionally well documented. Because of its simplicity and its clarity, it constitutes an excellent tool for illustrating operating system concepts.

The only drawback of the system lies in the fact that it requires its own dedicated hardware. If we wanted to let students experiment with the XINU networking facilities, we would have to provide each group of students with access to their private set of LSI-11's. We thus decided to build a XINU virtual machine to run on a host computer. This also gave us an opportunity to enhance the existing XINU facilities by improving the XINU user interface and by allowing the realistic simulation of a wide range of network topologies.

An initial implementation of our XINU virtual machine has been running for several months on VAX 780's and 750's under Berkeley 4.2 UNIX. Our virtual machine simulates in software all LSI-11 functions required by the XINU software.

Most of the work in porting standard XINU to the UNIX environment involved XINU kernel modifications and the writing of special I/O simulation code. Although the machine dependent parts of the code differ from the original XINU implementation, the XINU programmer's interface is compatible with the standard version. Since XINU is written in C, all XINU programs run in the native instruction set of the host computer, which results in a very lean and very efficient virtual machine. This approach has the major drawback of limiting our ability to implement in XINU any memory management scheme that would require the trapping of invalid address exceptions.

The remainder of this paper focuses on the most interesting aspects of our port, namely the XINU kernel, the I/O subsystem, the enhanced networking facilities, and the new XINU user interface. The two last sections review our experience with XINU as an educational tool and contain our conclusions.

2. THE XINU KERNEL

The XINU machine is simulated on the host machine by multiplexing one UNIX process into many XINU processes. A portion of the full UNIX process memory space is allocated for XINU and is used in the same way that standard XINU uses the LSI-11 memory. As in XINU, this memory space contains a text segment, an initialized data area, an uninitialized data area, and a free data area, from lowest to highest address respectively. The free data area is further subdivided into a heap space and a stack space, where the heap space provides allocatable storage and the stack space provides one stack per XINU process. The stack space grows from highest memory and the heap space grows from low memory.

In the rest of this section we describe how we ported the kernel to the VAX. This description is rather technical and detailed, and is not necessary for an understanding of the rest of the paper. A little background information on the VAX architecture will be necessary.

The VAX instruction set provides a sophisticated procedure call instruction. When this instruction is executed, the registers that the programmer (or compiler writer) wants saved are placed on the stack. Some registers, like the pc and processor status register, are always saved. A register called the frame pointer (fp) is automatically set to point to this save area, so that when the called procedure returns, the previous machine state can be restored efficiently. One of the saved registers is the frame pointer that points back at the previous context. For example, if A calls B and B calls C, the frame pointer points at the saved registers of B, one of which is the fp that points at the saved registers of A. Two other registers are always saved, the stack pointer (sp), and an argument pointer (ap).

As a result, saving the frame pointer, the stack pointer, the argument pointer and the status register of a XINU process in some table allows XINU to start executing some other XINU process on some other stack while guaranteeing that the first process can be continued at any time. To go back to the first process, the XINU kernel would have to load the fp, sp, ap and status register with the values from the table, and execute a return instruction. This instruction would cause the values in the stack save area that the fp points at to be loaded into the machine registers, thus continuing the process where it left off.

Here is a slightly simplified example of a context switch:

A UNIX alarm clock signal is received. An interrupt handling routine checks to see if it is time to preempt the currently running XINU process, which we designate process O, for old. If it is time to preempt process O, the handler calls a C function that selects a process N which will be the next to run on the CPU. The scheduler then calls a short assembly language procedure, passing it pointers to the process table entries for O and N.

This assembler procedure, *ctxsw* for "context switch", stores the current fp, ap, and sp, in the process table entry for process O. It also stores another variable, called __clk__stat, which plays the role of the status word, in the sense that it records how UNIX signals are to be treated. It then loads these same registers and __clk__stat with values it gets from N's table entry. The last thing *ctxsw* does is to execute a return from procedure call. This has the effect that the saved context area on N's stack gets loaded into the machine registers, and process N picks up where it left off. It is an odd use of the return instruction, since control does not end up going immediately back to the caller, O, but to N.

Let us now see what happens when a process completes. When the process was created, an initial stack frame was constructed in such a way that when the main procedure of a process completes, it "returns" to a system function which removes the process from system tables, deallocates its stack space, and then calls the scheduler to schedule another process.

The function that creates each process actually sets up two stack frames. The first frame has the arguments the process was passed when created, and pointers to an uninitialized frame below, to be used by the system cleanup functions referred in the previous paragraph. The second frame is constructed to look exactly as if the process had been interrupted and switched away from. When the new process is selected to run, the pc that is "restored" from this second frame points at the first instruction of the new process.

It is interesting to note that the only change required in the kernel was to modify one short assembly language procedure and the process creation function, and to make small changes in the format of the process table entries.

In our XINU virtual machine, we use an interval timer to time a variety of simulated hardware events. This means that we can't completely block out the timer signals which are analogous to clock interrupts. What we do instead is set __clk__stat to DISABLED. This means that when the signal handler is invoked, it can check __clk__stat to see whether context switches or other potentially dangerous operations are permissible, and it can increment I/O simulation delay counters.

3. THE I/O SUBSYSTEM

We ported the XINU I/O subsystem to the VAX, to allow students to write device drivers for simulated interrupt-driven hardware. The changes we introduced are minimal enough that the XINU textbook [Come84] provides sufficient guidance for programming projects, with only a small amount of additional information needed to describe interface conventions for specific pieces of simulated hardware. Some of the software described in the next two sections was still undergoing changes at the time of this writing.

3.1 Device-Independent I/O

One of the nice things about XINU is that it is fairly successful at encapsulating hardware-dependent parts of the code into a small number of routines and data structures. The same apparatus that shields users from having to know hardware details, also makes the job of porting the system easier, even to an environment where all hardware is simulated. In those sections of the code where it was necessary to diverge from the standard, we use code which closely parallels the original approach, so as to be able to use the higher level software without change.

3.2 Interrupt Handling

A good example of this is the interrupt dispatch code. In the original, all interrupt vectors point at the same dispatcher code. That dispatcher calls the appropriate handler, written in C. The dispatcher knows which routine to call because one of the fields of the vector process status word points into a table containing pointers to all the lower-half device drivers that do the actual work of responding to each interrupt. We imitate this arrangement by having the actual signal handlers set a variable with a number indicating the source of the signal, and then calling a dispatcher which uses this number in the same way that LSI-11 XINU uses the vector psw. This way we didn't have to change any data structures or the device driver initialization mechanism.

The actual signal generators and handlers are black boxes from the point of view of students, with only some fixed input and output parameters known. Writing code to cope with a set of these that might change presents many of the same design problems as does writing device drivers for real hardware.

3.3 The Virtual Disk and File System

The initial motivation behind putting in a disk simulator was to let students write scheduling algorithms that would treat I/O-bound and compute-bound processes differently. Other interesting projects might involve modifying the system to manage concurrent access to the same file by two different processes, implementing an alternative to unbuffered disk writes, or integrating the file system into a network.

From the point of view of XINU software, a disk drive is an array of 512-byte blocks, which must be accessed by using a particular protocol, and the accesses have certain delays associated with them. Our virtual disk is a large UNIX file. The format of this file is a structure containing: an initialized directory, a linked list of free index blocks (similar to i-nodes), a linked list of free data blocks, and a couple of text files and their associated index nodes, all conforming to XINU conventions.

4.2 UNIX has a system utility which allows a user to establish an interval timer for a given process. This interval timer sends signals to the process at a fixed rate specified by the user. The handler bound to this signal performs such functions as counting off time slices and access delays.

Access delays are achieved as follows:

1) The XINU process requesting file access is suspended.

2) The clock signal handler counts ticks for as long as a real disk would take to respond.

3) The requested data is moved (by UNIX) to or from the specified buffer.

4) The requesting process is put back on the ready queue, so it becomes eligible to regain the CPU.

Of course, while the requesting process is suspended, other XINU processes are running.

4. THE NETWORK FACILITIES

In standard XINU, software is provided with each PDP-11/02 so that they can be linked together in a ring network, through standard RS232 ports. We wanted to provide a software simulation that would allow users to set up their own network topologies connecting the UNIX processes that contain virtual machines. We also wanted to be able to be able to simulate such undesirable realities of networking as line noise and dead machines, in a controlled manner.

To maximize flexibility, the XINU network is simulated by a socket structure with a central server and clients, one client mapped to each XINU machine. The central server design localizes the control of hazardous conditions such as noisy links and dead machines. All communication is routed via the central server, with a source client sending data first to the server and then the server sending the data to the destination client. The communication lines are statically created at the boot time of the network, thus emulating physical hardware links.

The topology of the XINU network is set up through connection with the central server socket and subsequent initialization codes sent over the newly created socket. Each XINU communication line requires two sockets between client and server. The information used in the communication line initialization sequence includes the process id of the requesting machine (for signals), the id of the requesting machine (for routing), and the id of the destination machine (for routing). After the destination machine has connected, communication can commence.

The central server dispatches source client send requests, at which time the server reads the byte, interrupts the source client, sends the byte, and interrupts the destination client. These interrupts correspond to interrupt on output buffer empty and interrupt on input buffer full respectively. With this facility, the simulated hardware and the interrupt handler can be plugged in without affecting the higher network layers. The interface to the higher levels was still undergoing some changes at the time of this writing.

Through the central server, all sorts of test conditions can be simulated, including noisy lines and dead machines. With a simple random number generator, noise can be probabilistically entered in the data stream and machines can brought down.

Because of the complexity of a network system, we feel that an interactive central server with a debugging system would be desirable. The interactive server would be run in the foreground allowing user control over the network through interactive commands. For example, this facility would provide the capability of the adding noise to lines and bringing down machines deterministically instead of probabilistically and tracing the activity on the connections.

5. THE USER INTERFACE

Since XINU is written in C, the XINU virtual machine can be easily modified by changing the source code and recompiling it using the standard UNIX C compiler. Our implementation is strictly compatible with all the existing XINU system calls with the single exception of the command creating semaphores, *screate*(), which now accepts a second argument specifying a name for the semaphore being created. This naming allows easy semaphore tracing and process table printing.

The original version of XINU did not allow the loading of user code without recompiling the whole system. We enhanced XINU by adding a shell and an interactive tracing facility. Our new XINU shell allows the user to:

- directly load executable code without recompiling the whole system,
- display the status of all XINU processes,
- issue interactive system commands such as changing the priority of a process or killing it, and
- issue keyboard interrupts to kill or suspend the current process.

The tracing facility allows XINU users to trace preemptions, context switches and operations on semaphores. It can be called from the XINU shell or from within the program being traced.

6. XINU AS AN EDUCATIONAL TOOL

We wanted to give students homework assignments which would require them to understand something about the working of an operating system kernel. The first such assignment we gave consisted of having students modify the process scheduler to distinguish between I/O-bound and CPU-bound processes. We found that almost all of the facilities necessary to do this are already available in the XINU kernel code, and that the changes necessary were localized to a few procedures.

To teach network communications concepts, we provide each user with a number of XINU virtual machines, each of which is a UNIX process. These processes communicate with each other through sockets which simulate RS-232 ports. Hazards such as noisy lines and dead machines along the data path are simulated, so students get a realistic view of the requirements of communication software at the data link layer and at the network layer.

We have found that students tend to develop a much better grasp of crucial ideas when they have had some hands-on experience with system code.

7. CONCLUSION

We have presented a XINU virtual machine running on VAX architectures under Berkeley 4.2 UNIX. The key features of our implementation are

- a C language interface with user programs, allowing direct execution of all XINU system and user code,
- an improved user interface with an interactive shell and a debugging facility, and
- considerably enhanced networking facilities providing for the realistic simulation of a wide range of point-to-point network topologies.

An initial implementation of our XINU virtual machine has been running since January 1985. It has been used to teach operating systems concepts to more than one hundred and fifty students, without significant problems.

The major limitation of our design is its inability to implement in XINU any memory management scheme that would require the trapping of invalid address exceptions. Future work should include

- better monitoring functions facilitating the evaluation of the impact on system performance of any modification of the existing XINU policies,
- the port of our virtual machine to different host architectures and different UNIX environments, and

- a redesigned kernel that would allow the simulation of more sophisticated memory management schemes, including virtual memory.

A more radical departure to the existing XINU philosophy would be to use XINU to support a simple concurrent programming language like Occam.

REFERENCES

[Baba83a] Babaoglu, O. and F. Schneider, "Documentation for the CHIP Computer System," Technical Report TR 83-584, Department of Computer Science, Cornell University, Ithaca, NY, 1983.

[Baba83b] Babaoglu, O. and F. Schneider, "The HOCA Operating System Specifications," Technical Report TR 83-585, Department of Computer Science, Cornell University, Ithaca, NY, 1983.

[Come84] Comer, D. *Operating System Design: The XINU Approach*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

[Fabr83] Fabry, R. S. et al. "The TOY Operating System," CS 162 Notes Vol. I, Department of EECS—Computer Sciences Division, University of California, Berkeley, CA.

Device Drivers in a Multiprocessor Environment

Ed Gould

MT XINU
2910 Seventh Street, Suite 120
Berkeley, CA 94710

415/644-0146

{ucbvax,decvax}!mtxinu!ed

ABSTRACT

In the traditional UNIX* operating system, there is a pervasive assumption that a kernel process will not be interrupted in the midst of its task. In a multiprocessor system, where such a process may run on whatever processor happens to be free at the moment, this assumption breaks down. No longer is it the case that only device interrupts need be prevented in order to lock access to a data structure, nor is it reasonable to apply a global, system-wide lock whenever such a structure is to be accessed.

In this paper, the consequences of writing a device driver in a multiprocessor environment are examined. A number of issues are presented, including what locks are necessary and when they need to be applied, and how, for efficiency reasons, some of the algorithms of driver entries such as *ioctl* may be modified.

Introduction

The UNIX operating system as we have come to know it is implemented for a single processor computer system. A few forays into dual processors have been made, most notably at Purdue University where two VAX-11/780 CPUs were connected to one backplane, but these systems have not been general multiprocessors. In particular, they typically use a master-slave arrangement whereby only one of the processors may execute the UNIX kernel. This restriction is required by the fact that throughout the kernel is the assumption that kernel processes will not be preempted, and that they may only be interrupted to enter a device service routine.

Multi-microprocessor computers are now coming onto the market, and UNIX is a very attractive operating system for these machines. However, the single-thread assumption in the kernel is a severe hindrance to the implementation of UNIX in an environment where all of the processors are treated equally. Modifications to the kernel are required to remove this restriction, but device drivers remain quite similar to the standard single processor drivers. Some changes must be made, of course, to accommodate the multiprocessor environment, and they will be examined here.

---

* UNIX is a trademark of AT&T Bell Laboratories.

Single Processor Device Drivers

A device driver in a single processor system makes use of the single-thread assumption in order to gain exclusive access to the various data structures that it uses to perform its tasks. Activities such as adding to a queue of requests are clear examples of the need for exclusive access; if two processes attempted to add to the queue at once chaos would surely result. Other things the driver may do are more subtle. For example, certain devices contain read-once registers. If such registers need to be examined, exclusive access to the device is required. 1.

The algorithms of some of the entries to the driver assume a single-threaded execution as well. For example, the *ioctl* entries to many of the 4.2BSD drivers use a single structure for all instantiations of the routine. This implies that there may be not more than one user process executing an *ioctl* at any one time. In a multiprocessor, this becomes more of a disadvantage.

The Multiprocessor System

As an example of a real multiprocessor system, the Sequent Computer Systems Balance 8000 will be used, since that is the machine for which the author has developed device drivers. The Balance 8000 consists of from two to 12 CPUs, each with local cache, connected to a pair of Sequent-proprietary busses. The first ("Sequent") bus is a fairly usual sort of high-speed system interconnect. The second ("SLIC" — Serial Lock and Interrupt Control) bus handles device interrupts and provides the hardware assist necessary to guarantee interprocessor locking. Peripheral devices are also connected to both busses; memory is connected to the Sequent bus. 2.

Figure 1 shows the basic architecture of the Balance system. CPUs are mounted two per card, there may be from 0.5 to 16 megabytes of semiconductor memory, and one or more SCSI-Ether (SCED) and Multibus Adapters (MBAd) may be connected.

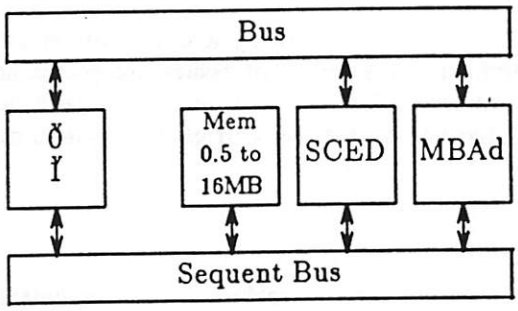


Figure 1 — Balance 8000 Architecture

The CPUs are normally used in a co-equal manner, *i.e.*, none is more important than any other. 3. In particular, when a device requests an interrupt, any CPU may be dispatched to handle it.

1. Devices such as terminal multiplexers contain examples of read-once registers. Often in a multiplexer, reading the "data input" register yields a flag indicating whether the remainder of the register contains valid data, a line number, and an input character. The act of reading the register causes the multiplexer to advance its state so that the next read yields the next input character.

2. Actually, memory is connected to both busses, but the SLIC bus is used only for control functions (*e.g.*, enabling or disabling a memory controller in an off-line manner, or determining if a controller is present on the bus).

3. It is possible to request that the system use fewer CPUs than are physically connected, but this is not typically done during normal timesharing use. It is also possible to designate a single CPU that will handle the interrupts from a particular device, but it is intended that this feature be used only as an interim step while converting a single processor driver to the multiprocessor environment.

The Multiprocessor Environment

When dealing with multiple processors, interlocking is necessary to keep the various processors out of each other's way. In the Sequent system, there are three levels of locks. The SLIC provides a set of hardware interlocks known as "gates." Using gates, the software provides both a simple test-and-set interlock called a "lock," and a fully queued, first-in-first-out semaphore system. 4. Programmers may allocate locks and semaphores as necessary.

Some of the traditional UNIX mechanisms may be replaced by some of these locking primatives. In the device driver realm, it is interesting to note that *sleep()* and *wakeup()* can be replaced by semaphores. There is a side-effect benefit of using semaphores in that processes can be conveniently scheduled in a first-in-first-out manner without additional complexity. (Of course, if *sleep* and *wakeup* are to be replaced by semaphores, the change must be made throughout the kernel, not just in the drivers.)

Multiprocessor Device Drivers

Designing a driver for the multiprocessor environment differs somewhat from the single processor case. The programmer must decide first which regions of the driver need to be interlocked, and then decide how many different locks are appropriate. Clearly, it would be possible to create a separate lock for each data structure that requires exclusive access, remembering to treat the device registers themselves as such a structure. However, there are typically several different structures, and creating a lock for each one is probably excessive. Second, the programmer must choose between *locks* and *semaphores*. Locks are faster and may be used whenever the requirement is only for mutual exclusion; semaphores are slower but provide more functionality.

Locks are usually the correct choice for most of the manipulations that are made by a driver. For most data, only mutual exclusion is required to ensure that the structures remain consistent and that work is done only once. Semaphores are useful for resource allocation where queueing is required.

The tradeoff between multiple locks for many data structures and one for the entire device balances throughput considerations against space. More locks may allow for a smaller elapsed time to complete a task since the lock required to complete the task is less likely to be busy. Locks consume memory, and although not much is needed for any individual lock, the total for a variety of locks can be noticeable. Also, using several locks may be more prone to programming errors than using one. The relationships between data structures are often subtle, and locking one structure may not be enough to maintain consistency. The efficiency question can probably not be answered except by measurement of running systems, but it seems intuitively true that there is no overriding reason to use multiple locks.

Debugging in the Multiprocessor Environment

Surprisingly, at least for most devices, debugging a multiprocessor driver is nearly as easy as debugging a standard driver. Most of the general algorithmic debugging can be done in a single-processor environment, simply by disabling all but one CPU. Once the basic driver is functioning, additional CPUs can be enabled to test the interlocking. No special tools or techniques are needed to debug normal multiprocessor drivers. 5.

---

4. The full details of the workings of the locks and semaphores will not be discussed here. The implementation provides a number of blocking and non-blocking functions to access them, but completely understanding the richness of the interface is not important to the topic of device drivers.

5. Good debugging tools always help the process, and in the development of the Balance system, Sequent developed some good, robust techniques and tools for dealing with their system. None of the multiprocessor features of these tools were needed in debugging the disk and magtape drivers developed by the author.

Algorithmic Changes

Some of the algorithms used in existing drivers are geared to the single processor environment. In particular, special functions implemented via the *ioctl()* driver entry often use a mechanism that is distinct from normal operations. The result is that only one process may access *ioctl* at a time, and that often normal operations must be suspended for a short time to allow the special function to complete. These limitations have not proven to be a problem in single processor systems, but in a multiprocessor they may be more trouble. It is not hard to change the implementation of the special functions so that they are handled in harmony with normal operations. The special operation may be packaged in such a way that it can be queued and processed in the same way as any other function. A flag may be needed to mark the package as special, and so that any special return information may be retained.

Conclusion

The multiprocessor environment will be an increasing part of the UNIX world as more multimicroprocessor machines reach the marketplace. Adapting the entire UNIX operating system to them is a very big job, but generating device drivers for them is not much more difficult than writing drivers for single processor systems. A few additional considerations must be taken into account, but they do not pose a major problem. The additional code required to support the interlocking necessary to keep the driver's data structures consistent is reasonably small.

PORTING THE AT&T DEMAND PAGED UNIX* IMPLEMENTATION TO MICROCOMPUTERS

Robert S. Jung

UniSoft Systems
739 Allston Way
Berkeley, California 94710
ucbvax!unisoft!bobj
May 7, 1985

Abstract

UniSoft Systems has been working with AT&T's UNIX System V Release 2 Version 4 (for the AT&T 3B20) to develop demand paged UNIX ports for microcomputer systems. An entirely new design of the memory management has been implemented in this latest release of the UNIX kernel.

This paper will cover:

- Memory management architecture prior to UNIX SVR2V4.
- The new well-defined interface to the UNIX memory management code.
- Architecture of the paging mechanism.
- Portability/non-portability of the code and ways to improve portability.
- How other sections of the kernel are affected.
- New kernel capabilities.

---

* UNIX is a registered trademark of AT&T.

1. Introduction

AT&T's UNIX System V Release 2 Version 4 implements an entirely new design of the memory management code in the kernel. The two major changes to the memory management are:

- implementation of DEMAND PAGING, and
- increased PORTABILITY of the memory management code.

AT&T's UNIX System V Release 2 (SVR2) and all of its predecessors (V6, V7, SIII, SVR1) have been based on a process swapping memory management scheme. Running large processes on swap-based systems is often impossible because this scheme limits the size of processes to the size of available physical memory. The UNIX SVR2V4 reimplementation of the memory management code uses a demand paging scheme, which allows the size of processes to extend to the full address space of the hardware architecture. Usually this virtual address space will be much larger than the actual physical address space.

One of the objectives of this redesign was to increase the portability of UNIX. Previous releases of the AT&T UNIX kernel did not define a standard memory management interface, but utilized other kernel functions to access the memory management hardware. The interface between the memory management functions of the kernel and other functions of the kernel is now well-defined in one section of code called the *memory manager*.

UniSoft has developed other demand paging UNIX implementations in the past. These schemes were either based upon UniSoft's own design or the paging scheme developed at UC Berkeley. Because the SVR2V4 design is more portable and offers similar functionality, UniSoft intends to base future demand paging UNIX implementation on this memory management architecture.

The source code for the AT&T 3B20 version of UNIX SVR2V4 was the starting point for UniSoft's SVR2V4 ports. So far, UniSoft has worked on three SVR2V4 UNIX ports. These systems are briefly described as follows:

System 1	Motorola MC68010 microprocessor Proprietary memory management unit (MMU) with two-level mapping segments in hardware registers and page tables in memory.
System 2	Motorola MC68020 microprocessor Motorola MMB memory management unit (gate array subset of the Motorola 68851 PMMU)
System 3	Motorola MC68010 microprocessor Signetics BMAC memory management unit (subset of the Signetics MAC)

Specific examples in this paper will refer primarily to the first system on which the early work was developed. The ports for the other two systems are based upon the work done for the the first system.

2. Previous Memory Management Architecture

For over three years UniSoft Systems has been engaged in the business of porting UNIX to microcomputer systems. This experience has included porting swap-based UNIX (Version 6, Version 7, System III, System V.1, and System V.2) systems to over 20 different memory management designs, and yet porting the memory management code for swap-based UNIX has never become a trivial task. The primary reason why porting the memory management code has not been more straightforward is that there has never been a true *memory manager*.

A variety of different sub-systems within the kernel manipulate and access memory management hardware and data structures themselves, rather than calling a memory management interface. Examples of kernel functions which directly accessed the MMU hardware in previous systems include: **sureg** (mapping user processes for context switches); **copyin/ copyout/ subyte/ fubyte/ suword/ fuword** (transferring data to and from the user's address space); **resume** (remapping user area); **vtop** (virtual to physical conversions); and **vtopage** (virtual to page table entry conversions). Each of these routines in the kernel would "peek and poke" the memory management unit (MMU) directly rather than using one well-defined interface. [8]

This type of software architecture made porting UNIX to a new MMU more difficult because, without a standard interface, hardware dependencies are not isolated in one section of code.

3. The Memory Manager

The design of the SVR2V4 memory manager is based on the idea that only the memory manager should access the memory management unit (MMU) hardware directly. Knowledge of how the MMU hardware works, the paging algorithms and data structures used to implement the memory management should be isolated from all other parts of the kernel.

3.1 Regions

The SVR2V4 memory manager design revolves around the concept of chunks of virtual memory called regions. A region is a chunk of memory that is contiguous in the logical address space. It is not bound to any particular logical address, but rather it is a logical unit used by the rest of the kernel to access and manipulate memory. All mapping of processes are done in base units of regions. The memory manager provides several functions to manipulate regions for various purposes, including: allocate, free, attach, detach, load, duplicate, grow, shrink, lock and unlock.

A region is represented by a *region list*, which is actually a list of page tables. Region lists correspond to the MMU hardware's segment tables (assuming that there are segment tables). Each valid entry in the region list points to a page table. Page tables are dynamically allocated from the pool of free pages and therefore are not necessarily contiguous. This save both time and space.

A disk block descriptor (DBD) table is allocated each time a page table is allocated for one of the entries in the region list. The DBD entry that corresponds to an entry in the page table contains information as to whether or not the page has a copy on disk. If the page does have a copy on disk, the DBD entry contains three pieces of information: (1) the page's *type* (no copy on disk, copy on swap, copy in file, demand zero, demand file); its *swap index* (which swap file copy is in); and *block number*. In Appendix 1, Figure 1 illustrates the AT&T 3B20 Region Structure.

3.2 Process Regions

Process regions (or *pregions*) are associated with individual processes, and are used to attach (or bind) a region to a particular address in the the user's virtual address space.

Each active process structure has a pointer to an entry in the pregion table. These entries contain four pieces of information on the particular pregion: (1) what *region* it is associated with, (2) what *virtual address* within the user address space the region is attached to, (3) *protection* information (A pregion can be read only or read/write), and (4) the pregion's *type*: text, data, stack, and shared memory (UniSoft has added a phys type for the **phys** system call. See section on portablity for a descripton on the **phys** system call.).

3.3 System Requirements and Limitations

This implementation is based upon hardware with a two-level mapping: segments and pages. Since region lists correspond to segment tables, the limit on the number of region lists is generally equivalent to the number of segment entries supported by the hardware. It is assumed that the hardware supports referenced, modified, and valid bits at the page level.

The 3B20 implementation assumes read/write protection at both the segment and page entry level.

4. Demand Paging

All previous releases of UNIX from AT&T have been swapping kernels. UNIX SVR2V4 is the first official AT&T release of a demand paging kernel. The distinction between the two schemes is that in a *swapping* kernel, a user process must be entirely resident in memory in order to execute. If not enough memory is available to run a process, the kernel will select a process already in memory and "swap" that process out to disk. In a *demand paging* kernel, only the instructions and data being referenced at any one time by a process are required to reside in memory. Two processes whose total size exceed the amount of physical memory could both execute at the same time by having only part of each process "paged" in.

4.1 Demand Paging Implementation of Regions

As described earlier, the memory manager presents a standard interface to the rest of the kernel, utilizing regions as the base logical unit in that interface. The rest of the kernel is not concerned with how regions are implemented. In the SVR2V4 demand paging kernel, the memory manager represents regions with tables of *pages*, where a page is a fixed size chunk of physical memory; for example, the 3B20 uses 2048 bytes for its page size. It is the responsibility of the memory manager to maintain and manipulate the pages in accordance with higher level function requests that manipulate the regions.

4.2 Page Maintenance

In this implementation, all of free physical memory (i.e., physical memory minus the kernel space and system data structures) is divided into fixed size pages. There is a page frame data table (PFDAT) which contains information on each of these pages. If the page is in use, then the information contained in the PFDAT includes where to find a copy of that page on disk. If the page is free, then there are links to other free pages. A hash list of all the free pages is maintained for the page allocation algorithm.

4.3 Page Replacement

In SVR2V4 UNIX, a special system process referred to as **vhand**, runs periodically to free pages of memory if memory is needed. For every active region, this process uses *reference* and *need reference* bits in the page table entry to mark each page of the region *eligible* or *not eligible* for paging out. After it has marked the pages of an entire region, then it checks if memory is needed. If so, it ensures that a copy of the page is on disk and then returns the page to the pool of free pages. Then the **vhand** process cycles to the next active region.

4.4 Page Faults

The SVR2V4 kernel classifies page faults into two types: *valid* faults and *protection* faults. The machine architecture generates exceptions in both cases and there are the trap handling routines **vfault** and **pfault** to handle each of these page fault types.

A valid fault occurs when the valid bit of a referenced page is not set. There are four possible reasons for getting a valid fault: (1) the page is *paged out*, (2) the page is *fill on demand*, (3) the page is *demand zero*, or (4) the page is *unrecoverable*. In cases (1) and (2) the page will be read into memory from the disk or swap file, or from the file system respectively. Case (3) will usually be uninitialized data (bss), in which case a page is allocated and filled with zeroes. In case (4) the user process will be sent a signal that will terminate the process with a fatal error.

A protection fault occurs when the reference page is valid, but the access type of the operation is not permitted by the protection mode of the page. A protection fault may have been caused by a *copy-on-write* page. If this is the case, then it clears the software copy-on-write bit, allocates a

new page, copies the contents of the original, and adjusts the page table entry to point to the new page. If the page was not a copy-on-write page, then there was a real error and a signal is sent to the user process that will terminate the process with a fatal error.

5. Kernel Functions Affected by the Region Interface

The following is a list of kernel functions and a brief explanation of how they use or are affected by the region interface of the memory manager:

exec — **Exec** unmaps the current process by detaching all of its pregions. For the process to be **exec**'ed it allocates regions if needed. If the region is shared it may not need to allocate a new one. Then it attaches regions for text, data, and stack.

fork — **Fork** duplicates regions of parent process and attaches them to child process. It will set up pages copy-on-write. A copy-on-write page is shared by the parent and child processes until one of them write to the page. Then a new page is allocated and the contents of the page is copied. Since most **fork**'s are followed immediately by an **exec**, this prevents a lot of needless page copying.

exit — Before a process exits it must unmap itself and release its region resources. It does so by detaching its pregions.

grow — When a process needs to grow the stack, it does a **findpreg** to find the stack region and then calls **growpreg**.

sbreak — **Sbreak** is just like **grow** except it works on the data region.

shared text — A text table is no longer used to keep track of shared text. Instead shared text is represented by regions with a STEXT type. Allocation, freeing, attaching, detaching, etc. are all managed using region functions.

shared memory — Similar to shared text, no special data structures are needed because shared memory is represented by regions with a SHMEM type.

phys — The **phys** system call is a UniSoft enhancement that allows the mapping of specific physical addresses into the user's logical address space. It is implemented with regions in a similar fashion as shared text and shared memory. Regions allocated by **phys** have PHYS region types.

6. New Kernel Capabilities

larger address space — The main reason for demand paging is to be able to run processes that are larger than the available memory.

new object file format — Allows load-on-demand files (also referred to as "413" files). This is accomplished by mapping files and paging them directly from the file. Non load-on-demand files must be loaded onto the swap device before they can be paged. The object file must have each section aligned (text, data, bss) on a disk block boundary.

multiple swap files — Allows dynamic reconfiguration of swap space spread across different disks. Because paging kernels tend to be more disk I/O intensive then swapping kernels, having swap on multiple devices can spread the load more evenly.

lock/unlock shared memory — In the paging system, users can request that a shared memory region be locked in memory (for better response), or unlock a locked shared memory region.

7. Portability

The portability issues associated with this kernel can be classified in four groups: (1) coding practices, (2) multi-processor implementation, (3) product compatibility, and (4) memory management assumptions. Most of the changes required to port the AT&T 3B20 UNIX SVR2V4 to a Motorola 68010-based super microcomputer system were isolated to the low-level details of the memory manager, and the emphasis in this section will be on problems in group (4). The other groups are mentioned briefly below.

Group (1) includes portions of code which were optimized either in microcode or assembly language for the 3B20 and were subsequently rewritten in C. Some examples of routines that were micro-coded for the 3B20 are **fubyte, subyte, swtch,** and **qswtch.** There are assumptions about size of data structures (i.e. page table entry is the size of an integer).

The code from which the port began included not only demand paging but also multi-processor capability [4]. This resulted in group (2) type problems such as a need to implement semaphores in the kernel.

Group (3) problems are caused by the commercial necessity of keeping UniSoft's product, UniPlus+, upwardly compatible. Before going to the Common Object File Format (COFF), UniPlus+ supported a different object file format [5], and it is important that UniSoft's customers be able to run their old programs. UniSoft has also enhanced the UNIX kernel with added features that had to be ported to the new demand page kernel, such as a faster system call interface, and the **phys** system call [10] which allows a user process to map specific physical addresses into it's logical address space. UniSoft has also integrated the 4.2 BSD IP/TCP networking code into it's SV2R2V4 Uniplus+.

The rest of this sections deals with group (4) portability issues.

7.1 Stack Growth

The code's assumption about the direction in which the stack grows caused serious problems in portability. On the 3B20 the stack grows from low memory to high memory. The Motorola 68000 family of microprocessors grow in the opposite direction (high to low). In the 3B20 code the stack regions are handled in the same fashion as the other regions (text, data, shared memory), but for the 68010 code each region routine must be able to determine if the region is a stack region and treat it differently. One example of how this affected the code is the routine that expands the page tables, **ptexpand**, which is passed a region pointer in the 3B20 code instead of a process region (pregion) pointer. When expanding the page tables for the stack for a Motorola 68010 system, the page table entries must be allocated from the end of the table to the beginning. Without the process region the type of region (text, data, stack, etc.) cannot be determined. UniSoft modified **ptexpand** to take a process region pointer instead of a region pointer as a parameter.

7.2 Software Bits in the Page Table Entries

Another assumption that makes the 3B20 code difficult to port is that certain optimizations were made utilizing unused bits in the page table entries. The paging algorithm needs its own software flag bits. These three software bits are *need reference*, *copy-on-write*, and *locked*. On the 3B20 not all of the 32 bits in the page table entries (PTEs) are used by the hardware. So in the 3B20 code the same 32-bit PTEs are used to store both the hardware and software bits. The assumption that there are unused bits in the PTEs does not hold true for other machine architectures. The systems UniSoft is porting to do not have 3 unused bits in their page table entries. (The first system ported to has only 16 bit page table entries.) UniSoft had to modify the code so that there are now two sets of page table entries for each page -- one for the MMU hardware and the other for the software bits. Every PTE access must use a special macro to access all the bits as if the software and hardware PTEs are actually one PTE. Appendix 1, Figure 2 illustrates the UniSoft UniPlus+ Region Structure.

7.3 Disk Block Descriptor Table Allocation and Access

In the 3B20 code, the DBD table is allocated immediately after the corresponding page table, and the DBD tables are the same size as the page tables. In the initial UniSoft port, additional software page tables are required since page table entries are 16 bits, and DBDs are 32 bits. Appendix 1, Figure 2 illustrates the UniSoft UniPlus+ Region Structure.

7.4 Kernel Access Through User Page Tables

When copying to or from the user's address space while in supervisor mode (i.e., **copyin** or **copyout**), the kernel should not get a supervisor level exception if the page in user space is not valid. The 3B20 hardware allows the kernel to map through the user's mapping. If the page being referenced needs to be paged in or if it is a copy-on-write page, then it is handled as if the user was accessing the page. If the page is really a bad address then an error will be returned.

The initial system UniSoft ported did not have this ability to use the user mapping in supervisor mode. If the page was not set up to be accessed, then a supervisor level exception occurred. Code had to be added so that each time the kernel accesses into the user address space, it checks permission and valid bits of the page table entry in software before attempting to do a read or write. If the page needed to be paged in or was a copy-on-write page, the kernel must take the appropriate corrective action before accessing into the user's address space.

7.5 Segment Tables in Registers vs. In-Memory Segment Tables

The 3B20 code expects both segment and page tables to be in memory. In the initial UniSoft port, the MMU hardware used segment tables in registers and page tables in memory. Because the segment tables are in registers instead of in memory, the initial system used context registers instead of system base registers to switch mappings. Context registers select which set of segment tables to use. System base registers point to which segment table in memory to map through. The original 3B20 code manipulating the SBR's had to be modified to use context registers instead.

7.6 User/Supervisor Protection

The 3B20 handles user/supervisor protection at the system base register level, but in many systems, user/supervisor protection is handled at the same levels (segment and page levels) as read/write protection. The code for that changes the segment and page read/write permission had to be modified to also set user/supervisor permission.

7.7 Low Level/Hardware Dependencies

Most of the hardware dependencies were isolated in routines and macro definitions such as: clearing address translation buffers (ATBs), trap and exception handling, and bit layout of page table entries. Some hardware can be more restrictive than others such as requiring page tables to begin at 256 byte boundaries.

7.8 Physio

On larger machines like the 3B20, virtual addresses are often used to do I/O. On many microcomputer systems, devices only perform I/O with physical addresses and do not use the vrtual address mapping. This causes problems for the **physio** routine which is used to transfer large amounts of contiguous data to and from block devices, because contiguous logical addresses in the paging system do not necessarily correspond to contiguous physical addresses. There are two ways to solve this problem. The first solution is to allocate a chunk of contiguous physical memory at startup time and map it into the kernel's address space. This chunk of memory would serve as a buffer for the block I/O transfers. The second approach is to force a process to remap its pages to correspond to contiguous physical pages when it attempts to do **physio**.

The advantage of the first solution is that it is easy to implement and provides efficient performance. Its limitations are that the maximum transfer size must be determined at system configuration time and that it reduces the available physical memory. The second approach is more complicated to implement and has less efficient performance, but the transfer size is only limited by the amount of physical memory and no physical memory has to be reserved.

UniSoft's current implementation uses the first solution.

7.9 Data Structure Size Independency

The page size, the disk block size, the segment table length (number of segment table entries), and the page table length (number of page table entries) were all parameterized and only required redefinitions.

8. Conclusion

The new well-defined memory manager interface has indeed made porting the UNIX kernel to a new memory management architecture much more straightforward. But while the increased portability of this new kernel is generally praiseworthy, there were some intrinsic problems. Some of the changes UniSoft has made help to alleviate these problems, and make the code more generally portable. Examples of these changes include placing the software page table entry bits in a separate structures instead of overlapping them with the hardware page table entries, rewriting micro-coded or assembly language routines in the C language, and changing the region code to handle stacks that grow from high memory to low memory.

The SVR2V4 UNIX memory interface concepts and paging algorithms are good steps in making the memory management portion of the UNIX kernel more portable. As the importance of demand paging UNIX systems continues to grow, new system architectures (as well as old) will be requiring ports of this UNIX kernel. Experience with the code and the architecture are also important factors in a system's portability, so as ports are done to more and more system architectures, the ease of this code's portability will surely increase.

Bibliography

[1] *AT&T 3B20S/A Computers - Model 2: Central Control Detailed Description*, June 1984.

[2] AT&T Bell Laboratories, *Design of UNIX Paging for the 3B20 Simplex Processor - Issue 1*, December 14, 1983.

[3] AT&T Bell Laboratories, *UNIX System Paging Feature Requirements - Issue 1*, January 5, 1984.

[4] Bach, M.J., Buroff, S.J., *Multiprocessor UNIX Systems*, AT&T Bell Lab. Tech. Journal, October 1984, pp. 1733-1750.

[5] *Chapter 11: The Common Object File Format*, UniPlus+ System, Release 2 Support Tools Guide Vol. 6 (Beta Copy), UniSoft Corporation, Berkeley, 1984.

[6] Jagels, D.P., *A Standard Interface to the UNIX Memory Manager*, AT&T Bell Laboratories, October 2, 1983.

[7] Jung, Robert S., Notes for talk on *MMUs & the UNIX Kernel*, EUUG Autumn 1984 Conference, September 1984.

[8] Lai, Clara S., Johnson, Chris Peer, *Memory Management Units and the UNIX Kernel*, USENIX 1984 Conference Proceedings, Salt Lake City, Utah, July 1984.

[9] *News - Paging*, UNIX System Support and Update News, AT&T Computer Systems Division - Computer System Support Center, Lisle, Illinois, April 1985, pp. 1-7.

[10] UniPlus+ System, Release 2 User's Manual Section 2-6, Vol. 2 (Beta Copy), UniSoft Corporation, Berkeley, 1984.

[11] *VMM - Virtual Memory Management Board - Preliminary Technical Description and User's Manual*, Cromemco, Inc., 1981.

APPENDIX 1

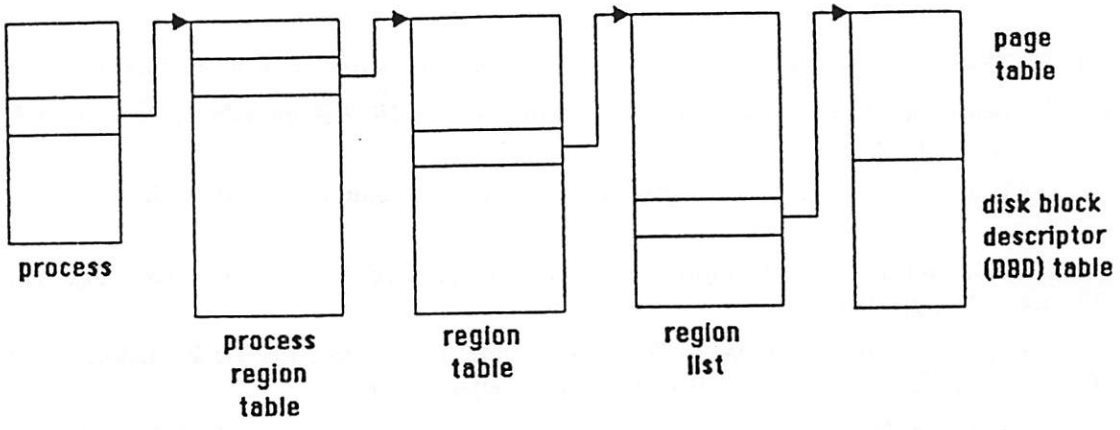


Figure 1: AT&T 3B20 Region Structure (note: adjacent page table and DBD table)

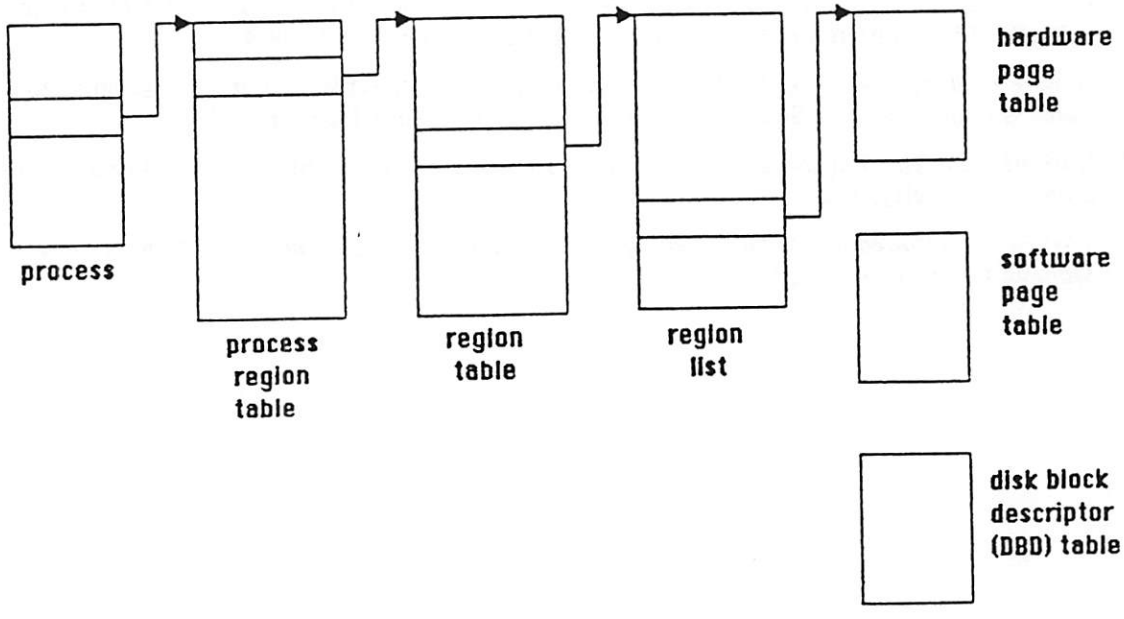
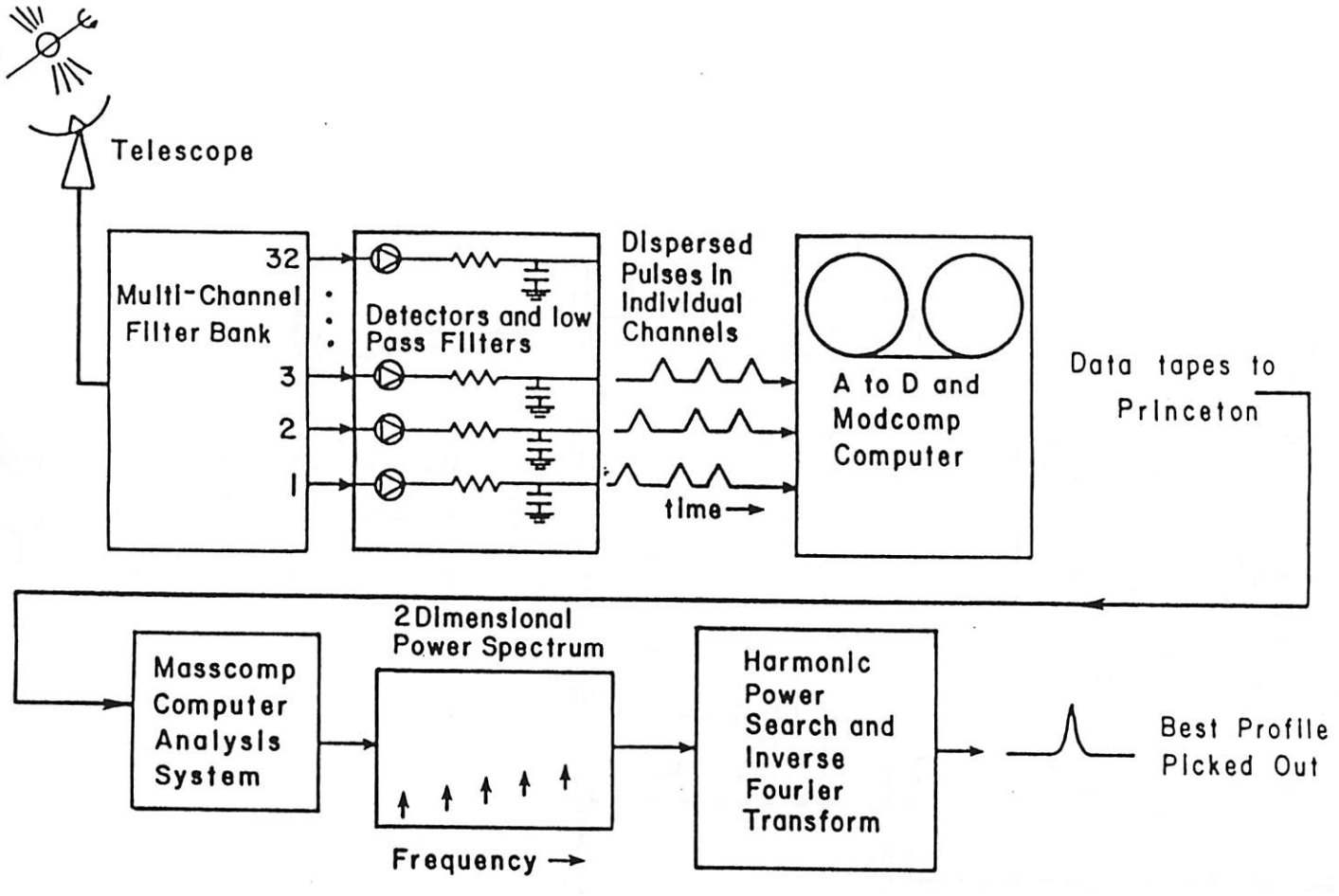
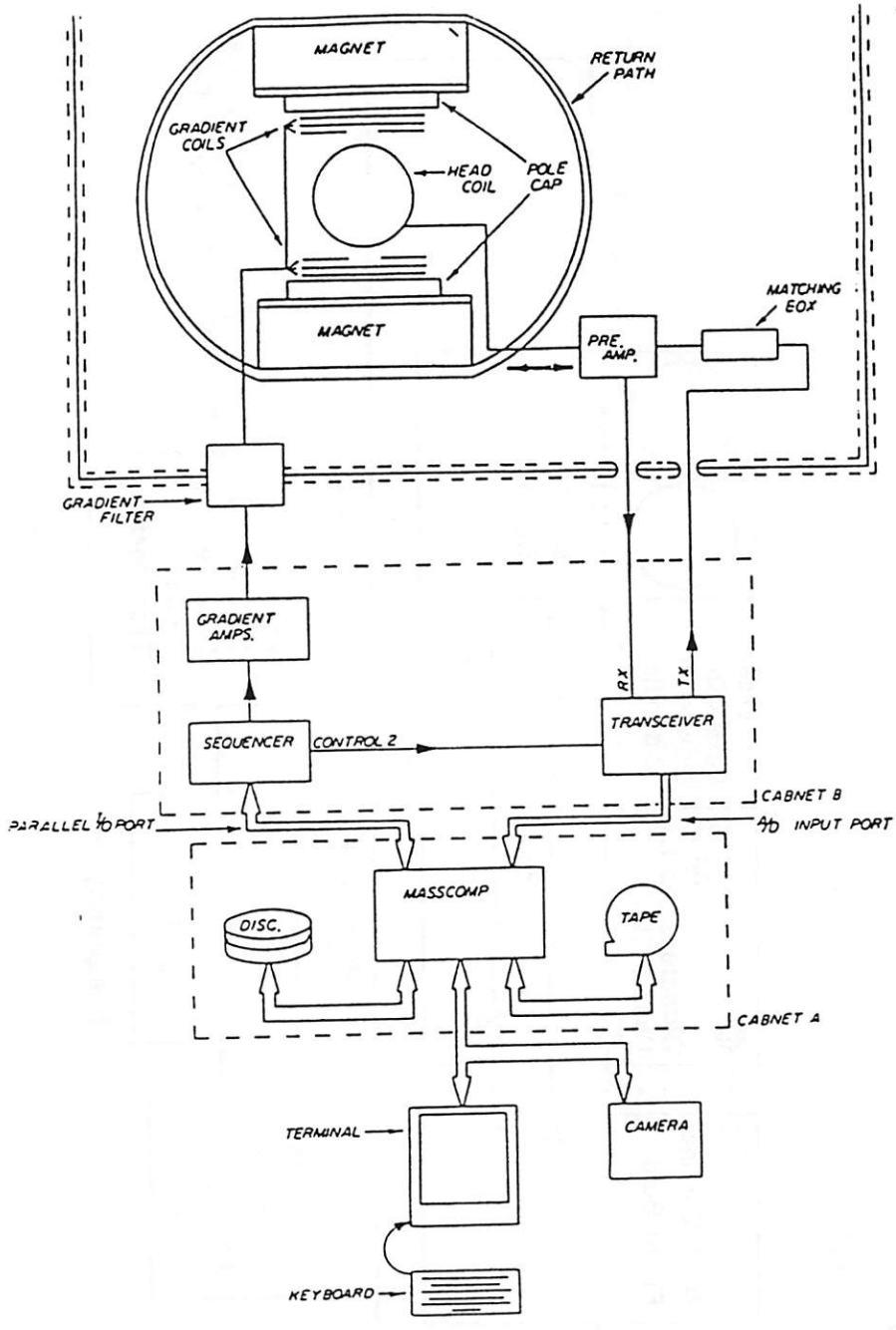


Figure 2: UniSoft Uniplus+ Region Structure (note: location of software page table and DBD table are determined by macro functions)



Princeton Apparatus and Analysis Algorithm



NMR IMAGING, INC.

A Capability Based Hierarchic Architecture for UNIX Window Management.

R. D. Trammell

Metheus

ABSTRACT

We discuss a formalism for window management in which the key components are:

* Strongly structured window objects.
* Hierarchic window organization.
* Inherited window Capabilities.
* Co-windows.

We show how this formalism is applied to work in progress at Metheus, a generalized UNIX window implementation in which a variety of plug-in window managers can cooperate to provide radically different application environments.

A Few Generalities

Windows can be decomposed into three disparate classes of functionality; text terminal, graphic display, and locator device. A well structured window abstraction should unify the treatment of these logical window devices (eg. which window(s) get the keyboard and mouse input). Some formal mechanism to regulate privileges and resources without imposing useless structure upon the model is also desirable. Although performance dictates some recourse to kernel code, we have attempted (without complete success) to limit kernel support for graphics to a device driver for the display subsystem and a window line discipline in the tty driver.

Lastly, we hope to support a wide variety of window environments. To this end, primitive window management functions normally performed by the OS or by privileged processes will reside in an intelligent front end processor to the display generator. Much of the ANSI CGI (aka VDI) compatible graphics interface layer is also resident there. This yields a degree of device independence in one direction, and compatibility with the CORE and GKS standards in the other. Information and responsibility is partitioned in such a way as to hide the details of window management and graphic device functionality from the kernel.

The Window Abstraction

We need a clear conceptual definition of 'window' before we can continue. A window is (for our purposes) a logical screen device, a rectangular subset of the physical screen. It owns attributes partitioned from other windows such as boundaries, a coordinate space, and a set of graphic Capabilities. It owns zero or more bitplanes which it may share with another window. A window's boundary may overlap other windows. A window priority mechanism protects non Co-windows (see below) from invading each other and controls their visibility. A window may be rendered invisible even though not covered by any other.

A window may have associated with it zero or more subwindows which share bitplanes, may overlap, and are each associated with a unique graphical context (eg. current drawing color

and position). Graphics are drawn in subwindows. A window is loosely associated with one or more processes. It possesses a logical keyboard and locator device to which it appears to have exclusive access, but in fact does not. A window is a text display surface which connected processes cannot distinguish from a 'real' text terminal.

A window's physical screen location and size may be dynamicly altered after which the window will retain it's transform but the clipping rectangle will reflect the new size. In the case of a move, the window contents are copied to the new location.

Windows Grow On Trees

Tree structures have been profitably employed in UNIX file and process organization. We shall apply them here as a vehicle for inheriting attributes, sharing resources and partitioning privilege. Windows occupy a hierarchy where each window (except the root) has a parent, and zero or more child windows. New windows are created by replication (analogous to the UNIX fork() system call). The child is called a Co-window. It copies the parent's attributes, sharing the parent's bitplanes, colormap, and display priority. Once created, it's attributes may be changed. It may acquire it's own bitplanes and relocate it's boundaries, etc. subject to Capability restraints. It ceases to be a Co-window when it alters the bitplanes allocated to it. Precisely then, a Co-window is one which shares bitplanes with another window. Co-windows make disjoint windows very cheap.

The Delegation Of Authority

Tree organization is used to structure window creation but it's main value lies in the control of graphic Capabilities. Every window inherits a Graphics Capability Set from it's parent, each element of which signifies some dimension of freedom in the window domain. A window may give up a Capability but may not acquire one. This is a classic Capability-based organization in which freedoms to manipulate a resource are associated with a particular descriptor of the resource rather than with the resource itself. A physical screen resource (area X depth) may be referred to by several window descriptors, each with a distinct Capability set. Some examples of Capabilities are; access to locator device and keyboard data when not 'top' window or when the locator is out of window bounds, power to modify attributes of other windows, power to grow or move beyond original bounds, and power to modify window display priority. Many Capabilities should never be passed to an application window. In fact, the graphics libraries may hide many of them from applications. They serve to create an environment which permits hierarchies of less and less privileged tools.

Text Terminal Emulation

In keeping with the concept of a window as a logical screen device, it also has text terminal features (ie. linewrap, scrolling, cursor addressing CR/LF, etc). Terminal emulation is bound to the window borders. There is one virtual terminal per window. If an application needs several independent virtual terminal areas, Co-windows can be used. An interesting issue arises with TERMCAP handling for windows, they cannot be represented by a permanent entry in /etc/termcap. Instead, libcurses will request one from the window manager who constructs it on-the-fly.

Terminal emulation is done in the intelligent display subsystem. It permits an application or window manager to inquire (through the window management primitives resident there) the row/column character address of screen coordinates, and to inquire what text is displayed between two such addresses. This supports inter-window cut/paste operations. Each virtual terminal is accessed through /dev/ttywN, a device which is in turn, associated with a window device /dev/wN by a window manager process.

Libraries And Window Managers, Concealing The Truth

An application interacting directly with OS drivers would see ANSI CGI graphics with generalized (but rather primitive) window handling functions alongside. We shall employ a window manager process to layer a more sophisticated and specific window environment over them. Applications will link to graphic libraries which present views of CORE, GKS, or a proprietary interface. The libraries hide details of communications with the window manager and window devices.

The window manager is a daemon process. It controls window resources at a level which requires infrequent, non-cpu-intensive activity. It is not concerned with graphic activity below the window level. As a user-space process, it insulates the OS from graphic concerns, is easy to develop and extend, and can offer the user a potential choice of window environments though the availability of several window managers. Note also that the top level window manager may provide more specialized window managers with privileged windows according to their needs. These lower level managers may support a GKS environment or emulate another vendor's application environment. Window level services (appendix A) are provided via socket connection to a manager. The manager also supplies interactive services such as window creation and destruction, job control, and cut/paste operations directly to the user.

Subwindow and CGI drawing primitives are display lists shipped directly to the display subsystem through window devices obtained from the manager. In most cases, a process running in a window has had the virtual terminal device associated with the window open/duped as it's standard streams by a window manager when the process was spawned. In other cases (eg. the manager itself), the process may have intentionally acquired and opened it's window and virtual terminal devices.

Working With init, The Window Definition File

Some special provisions must be made with init (the ancestor of all user processes) with respect to window 'terminals'. Init must spawn the window manager and a login process for the first (login) window. Login will inform the window manager when the login procedure successfully completes.

The window manager provides init with a login window. He will provide bare windows to applications on request, and shell windows to the user through icons or control characters after the first login as mentioned above.

A window definition file (watched by the window manager) specifies the attributes of predefined windows including bounds, colors, planes, Capabilities, and a process to run there. The specification defines when the window is to be created (after window manager initialization, after window login, or on user request for a predefined window).

OS Support For Graphics

Our design attempts to limit specialized OS intervention in graphics and window management. We are left with what appears to be the minimum consistent with reasonable performance. To this end, device dependent window management primitives such as window context switching and display priority control are relegated to the intelligent display processor. A window manger process(s) invokes these primitives using the OS primarily as a communications channel.

The OS supports a window tty line discipline which invokes a Display Subsystem Interface driver rather than the usual kl.c routines. The discipline ships clists tagged with window (minor device) number. On input, clists muxed by minor number are received from the DSI driver and sorted out to the correct processes. This clist traffic is copied through the kernel as with normal tty io. The DSI driver does it's DMA into kernel space. The discipline will not allow an open to complete until the corresponding window device has also been opened. If the window device is later closed, The window tty device will behave as if carrier had been dropped on a true serial

port. The discipline must notify the window manager (via socket) when a TIOCSPGRP ioctl is done on a window tty or upon last close of a window tty with pending close on the corresponding window device.

There is also a window device driver which invokes the DSI driver to transport purely graphic information. Here, the DMA is done directly to user space without copy through kernel buffers. The DSI driver notifies the window manager (via socket) on last close of a window with no corresponding open window tty. If the window tty is still open when the owner attempts to close the window itself, the window will be closed when the window tty is closed. The window driver must inform the window manager when a process with open windows exits. The window driver is responsible for the binding of processes to windows, sleeping and waking processes waiting on window or locator device events.

Applications view the locator device as part of the window mechanism, embedding requests for locator data in the window device opcode stream. There is no specific mouse device or driver visible to them.

The DSI driver handles the display subsystem link, performing the mux/demux of window and window tty traffic to processes without copy through kernel buffers. An interesting buffer handshake scheme permits the graphic libraries to control their own buffering strategy. Read/write requests do not block, they queue the DMA instead. As output items are picked off the queue, the user space buffer pages are locked in core while the DMA takes place. On input, the buffer pages are locked when the data become available until the transfer is complete. The driver sets the user space word preceding the buffer to indicate completion. The driver supports an ioctl which will block on a specified buffer address which must be in the queue. For example; The graphic library decides to maintain a ring of buffers. On output, it walks around the ring filling buffers and writing them to the window device. When it encounters a buffer without the done flag set, it issues the ioctl to block on that buffer done. When the ioctl returns, it resumes the walk. On input, the same algorithm is followed except in the cases which require blocking (eg. wait-locator-device-event). In such cases, the request is placed in the outgoing buffer which is written (non-blocking). A read for the input data (also non-blocking) is issued. Then an ioctl to block on the read buffer completion is done.

The Display Subsystem

The display subsystem consists of an intelligent front end processor driving a high performance color display generator. The FEP presents a CGI compatible interface on which we layer a terminal emulator and window server. The server talks to UNIX processes with messages wrapped in window/subwindow context, invoking the terminal emulator for window tty functions and the CGI layer for graphics.. It is loosely coupled to the workstation over a high speed parallel bus and provides the window handling primitives (appendix B) on which the UNIX window managers depend. This supports a variety of window management schemes, and reduces the bandwidth required for window level traffic. The window server fuses terminal emulation, locator device, keyboard, and graphic display into a single conceptual window device. The window hierarchy is formed at this level. Window display priority and the switching of window and subwindow context is handled here.

The window server associates a Graphic Capability Set with each window and applies these Capabilities to incoming requests. Decisions to route keyboard and locator device information are based on Capabilities and other window attributes.

Project Status

This work is the result of an effort to improve on our current Window Manager. As this is being written, we have a detailed design and are working to implement the window and graphics architecture described here on our current workstation hardware and a new display generator with intelligent front end processor. The display subsystem (and it's CGI compatible firmware) are

fully functional. We hope to have some of the UNIX window management code working in the next few weeks.

Conclusion

We have described a graphics architecture for UNIX workstations predicated on window trees, utilizing a classic Capability-based mechanism for defining window functionality and regulating privilege. We hope to demonstrate that the synergy between these two concepts forms a powerful tool for the construction of generalized multi-window environments.

Appendix A

Window Level Services Provided To A Client ProcesS

Some of these services are provided by the Window Manager daemon. Some are provided by the Display Subsystem window management primitives through an open window device. All but the first require a window descriptor and all are subject to Graphic Capability Set (GCS) control.

* Provide predefined window.
* Fork Co-Window.
* Modify bitplane allocation.
* Modify colormap.
* Modify bounds.
* Inquire GCS.
* Mask GCS.
* Modify window priority.
* Inquire window state, attributes, window and system planes used and configured.
* Inquire character row/col address of window coords.
* Inquire window text in row/col address range.
* Inquire a window's child window numbers.
* Inquire a window's parent window number.
* inquire TERMCAP string for window.
* make window invisible/visible.
* CSH style job control functions.

Appendix B

Display Subsystem Window Server Primitives

The Display Subsystem Window Server provides the following functions in support of UNIX Window Manager processes.

* Fork (replicate) window.
* Set window bounds.
* Modify window colormap.
* Modify bitplane allocation.
* Inquire GCS.
* Mask GCS.
* Free window.
* Create subwindow.
* Set subwindow bounds.
* Inquire subwindow attributes.
* Set subwindow writemask.
* Free subwindow.
* Select subwindow.
* Inquire locator device status immediate.
* Inquire locator device status on event (button and/or position change).
* Inquire row/col text address of window coordinates.
* Inquire text within text address span.
* Save/restore screen area to/from FEP memory.
* Inquire what child windows.
* Inquire what parent window.
* Rotate window priority stack up/down.
* Make specific window top/bottom.
* make window invisible.
* Inquire window attributes.
* inquire system attributes/configuration/current resources.
* Enable/disable auto screen-dimming function.

Mex - A Window Manager for the IRIS

Rocky Rhodes
Paul Haeberli
Kipp Hickman

Silicon Graphics Inc.
630 Clyde Court
Mountain View, California 94043
(415) 960-1980

ABSTRACT

The design and implementation of a window manager for the IRIS workstation is described. The graphics hardware and software of the IRIS provide relatively high level primitives for manipulating the display. Accordingly, traditional raster oriented operations for windowing are not appropriate, and a new approach is used. The primary goal is to provide multiple processes direct access to high performance hardware, without sacrificing performance or flexibility.

1. Introduction

The IRIS workstation consists of a 68010 processor with memory, disk, Ethernet, and custom graphics hardware. Multiple Exposure (mex) is a window manager that runs on the IRIS workstation. Mex controls the screen space allocated to each client graphics process and selects which process receives input. This functionality is split between a user process (mex) and the kernel. Mex provides a graphical user interface, manages screen space, and gives redraw events to clients when their windows are uncovered. The kernel manages input devices and maintains the graphics state of each process.

We considered many issues to maximize the performance of programs running in this environment. Graphics programs use a very fast and simple method to synchronize access to the hardware. As a result, mex introduces almost no performance degradation for windows which are fully exposed. Updating partially obscured windows is made efficient by hardware that can clip to a list of rectangles. Input events are sent directly to processes by the kernel to minimize overhead. A kernel resident terminal emulator simulates multiple ttys. Shared memory between clients and kernel is used for interprocess communication (IPC) and control.

A functional interface has been developed to allow programs to communicate in a convenient way with the window system and a collection of screen-oriented tools has been developed. This paper provides additional detail on the IRIS graphics system, the IRIS window manager, and some directions for future development.

2. The Graphics System

The graphics hardware of the IRIS is an alternative to raster-op hardware. While raster-op hardware provides primitives like drawline and bitblt in screen space, the IRIS provides object space operations which allow applications to draw in their own coordinate space. A block diagram of the graphics system is given in Figure 1.

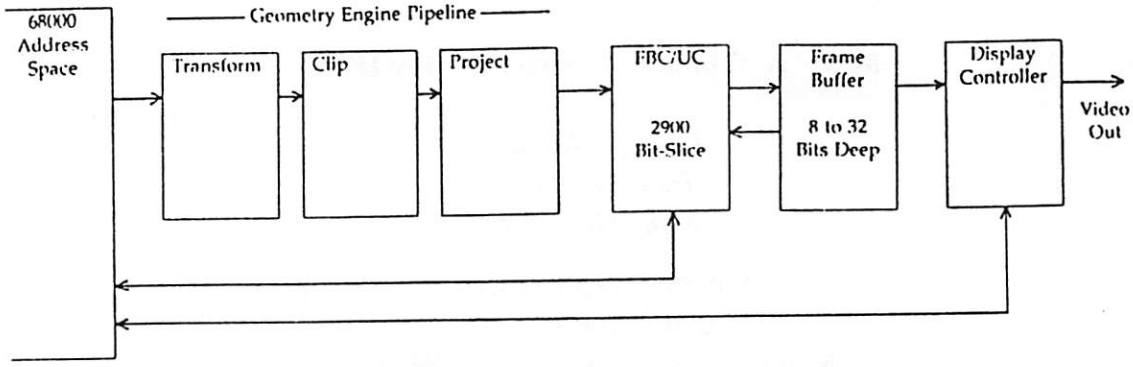


Figure 1. The graphics system hardware

The Geometry Engine (GE) pipeline is composed of 12 custom VLSI chips. This specialized hardware transforms, clips, and projects coordinates before objects are drawn into the frame buffer. The input to the geometry pipeline appears in the address space of each process running on the IRIS. Graphics commands are issued by writing streams of commands to this address. Figure 2 shows the command sequence for the **draw** primitive.

```
draw(x,y,z)
float x, y, z;
{
        register short *GE = GEADDR;
        *GE = GEdraw;
        *(float *)GE = x;
        *(float *)GE = y;
        *(float *)GE = z;
}
```

Figure 2. The draw primitive.

The first chips in the GE pipeline transform coordinates by multiplying them by a 4 by 4 matrix. The next set of chips clip coordinates against near, far, left, right, bottom, and top planes. The last set of chips perform perspective division and map coordinates onto a specific part of the screen. Commands (like GEdraw) come out of the pipeline with what are now screen space coordinates. The matrix multiplying section of the GE pipeline also supports a stack of matrices and operations on the top matrix. The top matrix may be loaded or multiplied by another matrix, and the stack of matrices may be pushed or popped. These operations allow applications to display hierarchical objects easily and efficiently.

The frame buffer controller (FBC) interprets the stream of commands coming out of the GE pipeline and issues commands to the update controller (UC), which draws lines and fills trapezoids in the frame buffer. The FBC consists of a 2903 bitslice with a writable micro-control store and a scratchpad memory. The FBC performs smooth shading and zbuffering of polygons, and also has the ability to feed data back to the 68010 by generating an interrupt.

The UC is able to draw stippled lines, patterned trapezoids, pixel streams, and character masks into the frame buffer, and can also read pixel data from the frame buffer. Character masks and patterns are stored in a font memory that may also be accessed by the 68010 and FBC. The display controller (DC) generates RGB video to drive a high-resolution color monitor by reading data from the frame buffer and passing these data through a color map.

This combination of custom hardware and host graphics library allows applications to be developed quickly. Figure 3 shows an example of a program segment which makes a cube out of four square sides. **Pushmatrix()** saves a copy of the top matrix, which is restored by **popmatrix()**.

Matrix operators **rot** and **translate** alter the matrix on the top of the stack.

```
cube() /* make a cube out of 4 squares */
{
        pushmatrix();
                side();
                rot(90.0,'x');
                side();
                rot(90.0,'x');
                side();
                rot(90.0,'x');
                side();
        popmatrix();
}
side() /* make a square translated 0.5 in the z direction */
{
        pushmatrix();
                translate(0.0,0.0,0.5);
                rect(-0.5,-0.5,0.5,0.5);
        popmatrix();
}
```

Figure 3. A graphics example

3. The Input System

All input devices such as mouse, dials, and tablet are treated in a consistent way. There are two major classes of input devices: buttons and valuators. The state of each type of device may be polled with calls of the form **getvaluator(MOUSEX)** and **getbutton(LEFTMOUSE)**. An input queue is also provided. Each event in the queue consists of a device number and a value. Individual elements are read using a statement like **dev = qread(&value)**. This read will block if there is no data in the queue. Presence of data in the queue may be tested with the function **qtest()**. A process may request that events from a device be added to the queue with a command of the form **qdevice(dev)**. If the device is a button (like **LEFTMOUSE**), an event will be added to the queue each time the button changes state. The value will be either 0 for up or 1 for down. If the device is a valuator (like **MOUSEX**), events will be added to the queue as the valuator changes.

4. The Window System

Implementing a window system has much more in common with operating systems than with graphics systems. Standard operating system concerns such as synchronization, resource management, and communication must be considered. Our window system is distributed between the kernel, a window manager process, and client graphics processes.

4.1 Synchronization

The nature of our present hardware requires that graphics commands be atomic. Working with this hardware requires a novel approach to handling synchronization. Figure 4 illustrates the synchronization problem. Process 1 was interrupted after partially completing a draw command to GE pipe. Process 2 then begins a similar draw operation. The complete state of our present graphics hardware cannot be saved in the middle of a command. The hardware will interpret Process 2's instructions as a continuation of the instruction stream of Process 1. The y coordinate of Process 2's draw operation will be interpreted as a new command and, barring incredible luck, the hardware will complain bitterly about an unimplemented instruction. If the hardware had been designed with multi-tasking in mind, and were interruptible at any time, no special synchronization support would be needed.

```
Process 1                    Process 2
*GE = GEdraw;
*(float *)GE = x;
                    *GE = GEdraw;          /* interpreted as a y co
   ord */
                    *(float *)GE = x;      /* interpreted as a z coor
   d */
                    *(float *)GE = y;      /* interpreted as a new co
   mmand (eeek) */
                    *(float *)GE = z;      /* also eeeek */
```

Figure 4. Interruption of the command stream.

Standard operating systems texts, such as [1] and [9], describe several solutions to the problem of synchronizing access to a shared resource. Messages could be used as the synchronization mechanism. The V system [2], among others, operates in this fashion. Figure 5 illustrates the general flavor of a graphics primitive using messages for synchronization.

```
draw(x, y, z)
float x, y, z;
{
      DrawMessage dm;
      dm.x = x;
      dm.y = y;
      dm.z = z;
      Send(dm, WindowManager);
}
```

Figure 5. Using message passing for synchronization.

All direct manipulation of the graphics hardware is performed by a window manager process. Synchronization of multiple clients is supported by message passing primitives. The window manager receives atomic messages (which represent graphics instructions from client processes) and then issues appropriate commands to the hardware.

One problem with this solution is satisfying the performance requirements of the system. Our graphics system is capable of performing roughly 100,000 3d transformations per second. Executing the **draw()** primitive described above should require no more tha
n 10 microseconds of the processor's cpu time to keep pace with the GE pipeline. Assuming 250 microseconds for the send operation [3], this operation is at least an order of magnitude too long.

A lower-level synchronization primitive used in many operating systems is the binary semaphore. The **p()**, or **wait()** primitive is used to request access to a shared resource. The user of this primitive is blocked until the resource becomes available. All other requesters are then blocked until the **v()**, or **signal()** primitive is called. Exclusive access to the resource is guaranteed between executions of **p()** and **v()**. Figure 6 illustrates the use of these operations.

The most obvious implementation of these primitives requires a system call for each **p()** or **v()** operation. Assuming 100 microseconds per system call throughput is limited to 5000 graphics primitives per second with this scheme.

Overall performance can be increased substantially in either message-passing or semaphore-based systems by amortizing the cost of synchronization operations over several calls. This can take the form of either explicit calls on the synchronization primitives by client programs (instead of the graphics primitives) or the use of display lists in graphics primitives. With a single synchronization operation spread over a suitably large number of primitive calls, synchronization cost becomes unnoticeable. The disadvantages of either of these two approaches are related to software complexities imposed on application programmers. Adding explicit synchronization

```
draw(x, y, z)
float x, y, z;
{
        register short *GE = GEADDR;
        p();                    /* wait for exclusive access to the hardware */
        *GE = GEdraw;
        *(float *)GE = x;
        *(float *)GE = y;
        *(float *)GE = z;
        v();                    /* signal end of critical section */
}
```

Figure 6. Use of p() and v() for synchronization.

primitives puts application programmers in the position of scheduling graphics resources for the entire system. Display list solutions usually require duplication of all graphics data structures, once in the application and once in the graphics system. Applications must be responsible for maintaining the consistency of two parallel data structures. Either way, unnecessary complexity is added to applications programs.

If suitable, albeit inefficient, methods are available for solving this problem, one approach is to speed up the execution of these synchronization primitives. A quick semaphore based on a page mapping trick was considered for the IRIS system. This operation (hack) required a single byte to be written twice for each graphics primitive. The p() operation would be implemented by writing a 1 to a special memory location. If the resource were not available, the page would not be mapped into this process's address space, and the resultant fault would cause the process to block until the resource became available. The v() operation would write a 0 to the same location, causing a fault if the resource were required by another process.

Preempting a process while it holds the resource prevents the system from scheduling any other process needing this resource. In this situation, a *convoy* of processes would form, all waiting for the high-traffic lock [4]. To prevent any single process from monopolizing the graphics hardware and to keep the system from dedicating most of its time to task switching, the scheduler needs to be made aware of this critical resource. The scheduler could examine the resource lock at any time and determine if the process were preemptable at this instant. A system that chooses to run the current process until it frees the resource will incur less overhead if this resource is frequently used.

This solution is equivalent to the one shown in Figure 6, but p() and v() operations are now only single machine instructions. This overhead is much easier to stomach than two system calls per primitive, but even the overhead of these two machine instructions was deemed too high. These two instructions would still use up approximately 20% of the 10 microseconds allowed per primitive to achieve the goal of 100,000 graphics operations per second.

Our solution improved on this technique. We implemented special graphics versions of p() and v() operators that impose almost no performance penalty. We tried doing this with magic [12], but practical considerations forced us to enhance the hardware. Our processor board has a single bit of state information (the magic bit), equivalent to the byte of memory used by the page mapping semaphore hack. There are actually two distinct addresses an application uses to access the GE port. Writing to one of these addresses has the side effect of setting this state bit, while writing to the other address clears the bit. The processor can read this state bit and enable or disable generation of an interrupt when the bit is cleared. The draw primitive using this scheme is shown in Figure 7.

This method is used in the following way to insure synchronization of multiple graphics processes. When scheduling a graphics process, the state of the synchronization bit is examined. If the bit is clear, then the last word moved to the GE port was the final word of a graphics command, and the system is free to interrupt the current graphics process and schedule the new one.

```
draw(x, y, z)
float x, y, z;
{
        register short *GE = GEADDR;
        *GE = GEdraw;
        *(float *)GE = x;
        *(float *)GE = y;
/*
 * After this instruction, the graphics pipeline is interruptible, so
 * write this value to the second address:
 */
        *(float *)(GE+1024) = z;
}
```

Figure 7. The IRIS's two address synchronization scheme

Otherwise, the current graphics process is in the middle of an uninterruptible graphics sequence. The system can then enable the synchronization interrupt and the graphics program will be interrupted as soon as it finishes its current command. Graphics commands will not be interrupted and system performance is not impaired.

Recent changes to the scheduler have been made to improve interaction. The standard scheduler has a fixed time quanta of one second. The scheduler was changed to have a variable quanta which was a function of the number of processes runnable on the run queue. So as the system gets busier, the scheduler gets called more often. The effect of this change is quite dramatic. Previously, if several cpu intensive programs were running, the interactive response of the window manager suffered. Now, every process gets approximately two time slices per second which is sufficient to keep the system interactive.

4.2 Resource Management

The synchronization scheme described in the previous section guarantees that access to the graphics pipeline is shared by client processes while insuring that graphics commands are atomic. Other system resources are shared among graphics processes. The state of our graphics system includes graphics attributes such as color, linestyle, and linewidth; a stack of floating point matrices; current character and graphics positions; current viewport; contents of a bank of memory used for storing fonts and textures; pixels stored in the frame buffer; entries in the color map; and input devices. All these resources are shared in one fashion or another by graphics programs run under our window manager. Some of these are virtualized to the extent that every process sees its own instance of the resource.

When a graphics process is scheduled, its matrix stack is restored, along with the current graphics and character positions, viewport and attributes. Thi
s can be done only after the state of the process currently using the graphics system is saved.

The ability of the FBC to feed data back to the 68010 can be used to capture most of the graphics hardware state. However, this path is slow compared to normal drawing operations. The 68010 is interrupted and then each word of data is passed from the FBC to the 68010 with a suitable handshake. Consequently, we avoid this path whenever possible. Copies of most state variables associated with the graphics attributes can be retained in the application's address space with little performance penalty. This technique is used to avoid using the feedback path for saving this portion of the graphics state. When a graphics process is scheduled, the operating system needs access to these state variables in order to restore the appropriate state to the graphics hardware. The IRIS graphics library uses a block of shared memory between the kernel and each of the graphics processes to store this information. With this mechanism, the graphics process keeps this state information up to date, and the kernel reloads this state into the hardware when a process that had been preempted is scheduled for execution.

Unfortunately, some state information is not so easily shadowed in software. The current graphics and character positions, and matrix stack are not directly available to software. These three pieces of state information are maintained only in the hardware and are fed back from the FBC to the 68010 for storage in the state vector of the graphics process when it is preempted.

This procedure of saving and restoring the graphics state adds to the time needed to perform a process switch between graphics processes. Approximately 1 millisecond is needed to complete a save-restore pair. Minimizing the time necessary for this operation is something to be considered in the design of future hardware.

Certain resources are allocated on a per process basis, rather than virtualized as described in the previous section. Font ram is handled in this fashion, because of the length of time involved in loading a font. In addition, pixels in the frame buffer are allocated to individual processes.

The complete contents of each window must be described somewhere to maintain consistency of the display as windows are moved or reordered. To do this, virtual (off screen) bit maps, a script of drawing instructions, or
a higher-level description of window contents may be used. Windows must also be mapped onto display surface to make them visible.

Pike's layerops [8] provide an elegant solution that maintains virtual bitmaps. The contents of each window is described by a bitmap image, part of which may not be visible. Bitblt is used to maintain the mapping to the display as windows are moved about. The Rainbow workstation [13] allocates bitmaps from special purpose graphics memory for each window. It has a sophisticated display controller to perform mapping onto the display.

Display lists or scripts of past commands may be used to update newly exposed window regions, but these schemes add complexity and extra overhead to normal drawing operations. VGTS [6] uses this scheme.

The solution we have adopted relies on the application to redraw its window contents from its own higher-level description. Mapping to the screen is performed by hardware that clips all drawing operations to a list of rectangular regions. This list of rectangles describes the visible area. We feel that this higher level solution is preferable. It is the only solution that handles resizing or reshaping of a window. Data is not duplicated unnecessarily, and the manner in which each window's image is described is left up to the application. We have found that adding the ability to repaint their windows adds little complexity to applications.

The IRIS color map is not managed. All graphics processes use one color map to interpret pixels in their windows. The current software allows any application to set color map entries. Cooperation is not enforced, but encouraged. Suites of cooperating applications are envisioned. A separate color map manager process could be developed to control this resource, taking requests for colors and returning appropriate color map entries.

4.3 Communication

The kernel, the window manager process, and the client processes all need to communicate with each other in this system. UNIX system calls could have been used for all user process to kernel communication and standard System V IPC could have been used for communication between the window manager and other user processes. However, the IRIS window manager uses neither of these communication facilities.

User to kernel communication is handled by a special graphics system call we have dubbed the graphics-ioctl, or grioctl. A single unix system call **grioctl(func,arg)** is used by user processes to invoke one of 100 different functions. A library of graphics related code is a part of our UNIX kernel. This library is also used in the version of V kernel [2] supported on our terminal products. The kernel graphics library (KGL) contains device handling code for the graphics devices, along with terminal emulation and input demultiplexing code. Grioctls are the standard interface to this library from user processes, and both KGL and grioctls are supported by our

graphics development group, not the operating systems group.

The interface between the KGL and the rest of the kernel has also been standardized and is supported by both our System V UNIX and V kernel. This allows the graphics group to support only one version of the KGL and only one version of the user graphics library, because the interface to the kernel i

s through our grioctls, not operating system dependent system calls.

Grioctls handle most, but not all of the user to kernel communication. As mentioned earlier, a copy of a large portion of graphics state is maintained in state variables in the user's address space. The kernel must have access to these variables to reload the graphics state of the process when it is scheduled. Both of our kernels, therefore, support a limited amount of shared memory between user programs and the kernel. This shared memory is primarily used for these state variables, but once the operating systems group made it available, the graphics group used it for a few other things as well.

The KGL maintains an input device queue for each graphics process. It keeps track of which devices are queued by each process, and which process currently should receive device events. When a new value arrives from an input device, the new state of the device is recorded in the kernel. Then an event is added to the queue of the current input procees, if this process queued this device. Having shared memory available has allowed us to implement a very fast version of our **qtest()** routine, where a user program inquires about the availability of input events in their input queue.

Shared memory and the input queues are also used for communication between application processes and the window manager process. We have implemented our own fixed-size, synchronous IPC mechanism between clients and the window manager by allowing a client to write a message in its own shared memory area, then enter a token in the window manager's queue. The window manager receives the token, and then asks the kernel to copy the message from the client's shared memory area to its own address space. After acting on the request, the window manager replies to the message, unblocking the sender. Grioctls are used for the system calls needed in these exchanges, and this limited IPC scheme is supported in our graphics group.

5. The Window Manager Process

Management of screen area, selection of input focus, allocation of kernel text windows, and interaction with the user are handled by the window manager process. When the window manager is started it registers itself as *the* window manager. This will cause it to receive messages from client graphics programs requesting window manager service. Messages from clients may request window creation, movement, or resizing. Other messages request that a window be exposed or hidden.

A user of the system interacts with the window manager using pop-up menus to move, resize, or destroy windows. The pop-up menus are also used to create new shells, and to select which window should receive data from the keyboard and other input devices.

Screen space is managed by keeping track of the display area allocated to each process as a list of rectangular pieces. Client processes or the user may ask that the arrangement of windows on the screen be changed. Mex updates its own piece structures and uses a grioctl to copy a new set of rectangles into the state vector of each window that has a new mapping to the display screen. Next mex uses another grioctl to insert a **REDRAW** event in the input queue of each window that has received additional exposure as a result of this rearrangement. This event indicates to the client that it should redraw the contents of its window.

The background of the display is drawn in a special way. A graphics program may register itself as the background drawer. It gets all the screen space not used by other windows. This process will receive a redraw event whenever any window is moved or destroyed over the background. This capability has been used to make textured backgrounds in addition to dynamic

backgrounds — one such background program fades windows out when they are moved from one place to another.

Input focus may be selected by the user. When this is done, mex uses a grioctl to tell the KGL that input events should go to the designated window. Clients can arrange to receive an event in their input queue when input focus is taken away or returned by queueing the device **INPUTCHANGE**. Certain events are queued for the window manager itself, no matter which window has the input focus. Using these *reserved* events, the user is able to interact with the window manager to rearrange, resize, destroy, create, or attach to other windows on the screen no matter who has the input focus. Button events may be bound to any of the actions above when the window manager process is started. Modifying a configuration file allows a user to customize her environment. This facility has been used to make mex work with a digitizing tablet instead of a mouse.

A common problem with high-resolution graphics systems is the performance of the text only windows. To support high performance textports we chose to put textports in the KGL. The KGL supports up to 10 text windows which emulate ascii terminals. These text windows are accessed by special files ttyw0 through ttyw9 in /dev. Terminal emulation in the kernel allows shells to print text to the display more efficiently than would be possible with the pty approach, at the expense of some flexibility. When the user indicates that a new shell should be created, mex uses a grioctl to obtain the number of a free kernel text window. The window number n that is returned corresponds to the device in /dev/ttywn. Next, after allowing the user to sweep out an area, a window is created on the screen, and a c-shell is started on the designated device.

6. Doing Graphics in a Window

Before a graphics program can draw, it must describe the screen space it requires. Programs may need a window with a fixed aspect ratio or size, or a window in a specific place. It may not need any screen space at all if it modifies only the color map or uses the geometry engine to transform coordinates. To make this description easy, a functional interface is provided which lets a program describe what it wants and get what it needs. A collection of qualifying functions may be called before the window is requested. **Keepaspect(x,y)** specifies that the aspect ratio should be fixed. **Prefsize(x,y)** describes the preferred size of the window. Other functions may be used to say that no screen area is needed, to describe minimum and maximum sizes that a window may be, or make a window that may be resized only in discrete steps like textports.

```
main(argc,argv)
int argc;
char **argv;
{
        short dev, val;
        int mx, my;
        keepaspect(3,2);   /* get a window with an aspect ratio of 3 to ? */
        minsize(75,50);
        getport("cube");
        qdevice(MOUSEX);   /* make mouse position go into the queue */
        qdevice(MOUSEY);
        perspective(300,3.0/2.0,0.01,1000.0);  /* set up the viewing matrix */
        translate(0.0,0.0,-4.0);
        while(1) {
                color(0);     /* draw a cube rotated by mouse x and y */
                clear();
                color(7);
                pushmatrix();
                   rot((mx-(XMAXSCREEN/2))*0.2,'x');
                   rot((my-(YMAXSCREEN/2))*0.2,'y');
                      cube();
                popmatrix();
              do {              /* read all the items in the queue */
                   switch(qread(&val)) {
                           case MOUSEX: mx = val;
                                   break;
                           case MOUSEY: my = val;
                                   break;
                           case REDRAW: reshapeviewport();
                                   break;
                   }
                } while(qtest());
        }
}
```

Figure 8. Graphics in a window

After these qualifications are given, a window is obtained by a call to **getport("name")**. This function forks to free the invoking shell, interacts with the kernel to get access to the graphics hardware, sends a message to the window manager, and waits for a reply. The window manager receives a token in its input queue saying that this process requests a graphics window. After interacting with the user for any unspecifie
d parameters (maybe letting them position the window on the screen), a reply is sent to the requesting process.

The process can determine the size and location of its window by the functions **getorigin(&x,&y)** and **getsize(&x,&y)**, if this information is needed. The application is now free to draw in its window.

If a window is moved, resized, or exposed, a redraw event will be added to that window's input queue by the window manager. It is assumed that all programs will at least occasionally read tokens from their input queues. The input queue was chosen for this purpose instead of UNIX signals. The signal method suffers from the problem that it may not be a convenient time to redraw when the signal is received. This forces the catcher to set a global flag which is tested by the program. A function **qtest()** may be used at any time to see if there is data in the input queue. Figure 8 shows a complete window manager program which makes the mouse control the

rotation of a cube. It requests a window no smaller than 75 by 50 pixels with an aspect ratio of 3 to 2. It then gathers input data and draws the cube.

Several tools have been developed to run in this environment. These include a screen magnifier, color editor, color map interpolator, mouse driven text editor, font editor, clock, and a tool to display image files on the screen. Graphics programs immediately put themselves in the background and interact on the screen. This means that one shell is all that is really needed to start a suite of tools.

7. Scary Things that go Bump in the Night

A number of complexities made the development of the window system more difficult. These include the lack of a true hardware cursor, support for double-buffered operation, and the capability of feeding data back from the end of the geometry system. In addition, the user and kernel graphics libraries had to run under two entirely different operating systems: UNIX System V and the V kernel.

8. Future Directions

There are many areas for continuing development. A manager of font ram will keep the most recently used fonts loaded into the hardware. Management of color map space is also needed. Support for popup menus, dialogue boxes and cutting and pasting between applications is under development. More general IPC under UNIX would be nice —- we would like to be able to use the window manager dynamically to connect programs together in building block fashion. This, for instance, would enable a paint program to get its brush shape from a generic shape editor, while the current texture and color come from a tool which manages textures or colors. The environment should also support the creation of dynamic documents. These will be books of interactive pages composed of networks of communicating processes [5]. Finally, we would like to see the system and tools evolve to the point that conventional shells will only be needed for system administration tasks.

9. Conclusions

The current system supports high performance graphics in a multi-window environment. Each client's view of the hardware is independent of whether the window manager is running. A macro package allows direct access to the graphics hardware for maximum performance. Client programs can draw lines at rates exceeding 100,000 vertices per second. A consistent method (the input queue) is used for all input, including redraw notification and IPC.

10. Acknowledgements

Many people contributed to this project. Jim Clark and Marc Hannah developed the geometry engine pipeline. Many thanks to Kurt Akeley, Peter Broadwell, Tom Davis, Mark Grossman, Herb Kuta, Steve Locke, Henry Moreton, and Gary Tarolli, who developed the geometry engine pipe adapter, frame buffer controller, update controller, bit plane boards, display controller, 2900 microcode, and the IRIS graphics library.

11. Bibliography

[1] Ben-Ari, M., Principles of Concurrent Programming, Prentice-Hall, 1982.

[2] Cheriton, D.R., and Zwaenepoel, W., "The distributed V kernel and its performance for diskless workstations", SIGOPS Operating Systems Review (ACM), 17(5) July 1983.

[3] Cheriton, D.R., "An Experiment using Registers for Fast Message-Based Inter process Communication", ACM Operating Systems Review 18(4), 1984.

[4] Date, C.J., An Introduction to Database Systems, Volume II, pp. 137-138, Addison-Wesley, 1983.

[5] Kay, A. and Goldberg, A. "Personal Dynamic Media", Computer, 10(3), March 1977.

[6] Lantz K., Nowicki W., "Structured Graphics for Distributed Systems", ACM Transactions on Graphics 3(1), pp. 23-51, 1984.

[7] Newman, W.M., and Sproull, R.F., Principles of Interactive Computer Graphics, 2nd ed. McGraw-Hill, 1979.

[8] Pike, R., "Graphics in Overlapping Bitmap Layers", ACM Transactions on Graphics 2(2), 1983.

[9] Silbershatz, A., Peterson, J., Operating System Concepts, Addison-Wesley, 1982.

[10] Silicon Graphics, IRIS User's Guide, 1984.

[11] Teitelman, W., "A Display Oriented Programmer's Assistant", Proc. 5th International Joint Conference on Artificial Intelligence, 1977.

[12] Weller, T., Science Made Stupid, Houghton Mifflin Company, 1985.

[13] Wilkes, A.J., et. al., "The Rainbow Workstation", UCCL, Cambridge, UK. 1983.

Windows for UNIX† at Lucasfilm

Michael J. Hawley

Samuel J. Leffler

Computer Division
Lucasfilm, Ltd.
P.O. Box 2009
San Rafael, CA 94912

ABSTRACT

The most successful UNIX-based window system to date has been that constructed for the *Blit* terminal. The Lucasfilm window system draws heavily on experience gained from the Blit, and incorporates ideas from other systems as well. Unlike the Blit, however, the Lucasfilm window system was built for a single-processor environment in which the graphics hardware is tightly coupled with the main processor. This paper discusses aspects of the bitmapped graphics interface with multiple, asynchronous overlapping windows under a 4.2BSD UNIX system (in this case, a 68010-based Sun workstation™).

Introduction

In the Lucasfilm window system, *windows* and other resources for interactive bitmapped graphics are available to C programs as just another library resource, much like *stdio* routines. Minimal kernel support is required. Graphics programs are written using the elegant *bitblt* abstraction formalized by [4]. This is augmented with an input handling style that can be both polled and event-driven. The graphics library includes a rich variety of user interface tools (e.g., routines for menus, graphical potentiometers, and text handling), as well as support for a variety of input devices such as keyboards, mice, touch screens, tablets, and special consoles. Applications write directly to the screen, using a global data base to calculate visible portions of a window. A window manager process and an ANSI terminal emulator provide a basic user interface for manipulating windows.

From a software engineering standpoint, the approach described here is similar to that used in the Blit. The major difference is that, in our system, applications typically reside entirely on the main processor — the Sun — while in the Blit, programs are split across a (slow) communications link. There are significant tradeoffs to be considered here. A major advantage of keeping everything on the main processor is that the rich arsenal of UNIX resources is still conveniently available to applications authors. Usual system facilities all work (e.g., the "Standard I/O" library, pipes, and networking), as do ordinary debuggers which operate equally well on interactive graphics programs. It is not necessary to develop special protocols to link graphics programs with the rest of the operating system. Application programs can be compact and understandable. Furthermore, network access under 4.2BSD permits applications to be divided over several processes on several machines if desired. Graphics interfaces running on the local workstation have been connected to processes controlling powerful digital audio and graphics machines, for instance. In the Blit's favor, a Sun-like machine is still comparatively expensive. More importantly, the quick, interactive *feel* of a Blit, the result of having a processor dedicated to managing

† UNIX is a Trademark of Bell Laboratories.

the "interaction box," is challenging to match on a single-processor Sun workstation. We have worked hard to keep interaction as fluid as possible within a 4.2BSD environment.

Section 1 of this paper describes the origin of the system and the motivation behind the design. Section 2 reviews the fundamental ideas used in the graphics portions of the system and section 3 presents components of the user interface *toolkit*. Section 4 describes parts of the window management system which are of interest. Section 5 discusses design and implementation issues. Section 6 summarizes current and future work.

1. Background

Our window system grew out of a need for a reasonable programming environment for the Sun workstation. In particular, the Digital Audio group at Lucasfilm was developing systems which used a Sun terminal as the basic user interface in conjunction with custom hardware consoles. Unfortunately, the window system distributed by Sun at the time failed to satisfy some basic requirements:

- simple things must be simple,
- hard things must be possible, and
- one must not be required to understand the entire system in order to do simple tasks.

Regarding the first point, the infamous "hello world" test immediately comes to mind. That is, how difficult is it to construct a program which creates a window on the screen, prints the message "hello world" in some font, and exits at the touch of a mouse button (say). Our encounters with the Sun software indicated that it took an experienced programmer more than a day to digest the Sun documentation in order to compose a tricky 2-page program which carried out this simple task.

Given our disinterest in existing Sun software, we looked for alternatives. Alas, at the time, the Blit programming environment, still the best UNIX window system we know of, was unavailable, as was the CMU system of Gosling and Rosenthal. Besides, the Blit software was optimized for a two-processor Version 8 UNIX world and hence not directly applicable to our environment. Still, we felt it would be productive to build on the strong concepts in the Blit design, and retain a style that would be familiar to programmers with Blit experience. So, we decided to create a Blit-like environment which took fullest advantage of available UNIX resources. To minimize the job and avoid losing all touch with potential Sun software development we chose not to completely replace the Sun windows. Instead, initial versions of the window system ran on top of Sun software, allowing Sun applications to coexist with our own. As users began to use our homegrown software exclusively, we started to replace and delete parts of the underlying Sun software. Today we have a system which uses only the lowest Sun layer —the *rasterop* and window-clipping functions. Retaining the rasterop level preserves device independence. The inclusion of the Sun clipping facilities was due to time constraints. We ultimately believe the management of overlapped windows is intimately tied to the basic rasterop function, as described in [3], but to implement such a system would have required replacing *all* the existing Sun software, more than we had time or desire to do.

2. Basics

The *bitmap/bitblt* abstraction has been described extensively elsewhere [1,4]. Many stylistic conventions come directly from Pike. Following is an overview of our implementation for the Sun world, covering essential data structures, arithmetic, drawing primitives, and user input resources.

2.1. Data Structures

We use these principal data structures:

```
typedef struct { short x, y; } Point;
typedef struct { Point o, c; } Rectangle;
typedef struct { Pixrect *pr; Rectangle r; int onscreen; } Bitmap;
```

A Point labels a location on a cartesian grid, with upper-left corner (0,0) and lower-right corner (XMAX,YMAX). Note that a Rectangle is given by two Points (o, upper-left *origin* and c, lower-right *corner*). We find this is cleaner than the location/size abstraction. To refer to sizes we use macros like:

```
#define XS(r)  (r.c.x-r.o.x) /* horizontal size */
#define YS(r)  (r.c.y-r.o.y) /* vertical size */
```

A Bitmap is a rectangular array of 1-bit pixels which represents an image. Here, pr is a pointer to the basic Sun bitmap-like object which holds the data contained in the bitmap[1]. If onscreen is non-zero, the bitmap represents an on-screen image, otherwise the image is off-screen. There is no distinction for the programmer between on-screen and off-screen images, and in fact, only the Rectangle member of the Bitmap structure is ever accessed directly. There are routines for reading and writing Bitmaps, and a declaration macro for including them in C source:

```
Bitmap *b; Rectangle r; char *file;

b = balloc(r)              /* allocate a Bitmap of size r */
    bfree(b)               /* free b */
b = ReadBitmap(file)       /* read a Bitmap from file */
    WriteBitmap(b,file)    /* write b to file */

#define DeclareBitmap(name, xsize, ysize, bits) ... /* static declaration macro */
DeclareBitmap(smiley, 16, 16, smiley_bits);
short smiley_bits[] = { ... };
```

2.2. Arithmetic

Lacking niceties like operator overloading, we use arithmetic primitives, making frequent use of the fact that C structures can be assigned, passed by value, and returned by functions:

```
Point p;
p = Pt(a, b)           /* construct and return the Point (a,b) */
p = add(a, b)          /* p.x = a.x + b.x; p.y = a.y + b.y; */
p = sub(a, b)          /* p.x = a.x - b.x; p.y = a.y - b.y; */
p = mul(a, b)          /* p.x = a.x * b.x; p.y = a.y * b.y; */
p = div(a, b)          /* p.x = a.x * b.x; p.y = a.y * b.y; */
eqpt(a, b)             /* a.x == b.x && a.y == b.y */

Rectangle r;
r = Rect(ox, oy, cx, cy)/* construct a rectangle */
r = Raddp(r, p)        /* r.o = add(r.o, p); r.c = add(r.c, p); */
r = Rsubp(r, p)        /* r.o = sub(r.o, p); r.c = sub(r.c, p); */
eqrect(a, b)           /* eqpt(a.o, b.o) && eqpt(a.c, b.c) */
```

The important thing to note is that Points and Rectangles are treated as atomic objects; structure members and pointer passing are used only when necessary.

[1] The pointer should really be a union of a *pixrect* and a *pixwin*, but this would disallow static initialization of bitmaps.

2.3. Drawing Routines

Drawing routines mimic the form of an assignment statement, in which a bit-source is drawn ("assigned") into a destination `Bitmap` at some location, using a boolean `Code` that specifies how the drawing is to be done. Basic `Codes` (and their Sun equivalents[2]) are:

```
#define F_STORE   PIX_SRC                    /* dst = src */
#define F_OR      PIX_SRC|PIX_DST            /* dst |= src */
#define F_CLR     PIX_DST&PIX_NOT(PIX_SRC)   /* dst &= ~src */
#define F_XOR     PIX_DST^PIX_SRC            /* dst ^= src */
```

Basic drawing primitives are:

```
Bitmap *b, *dst, *src;
Point p, p1, p2;
Rectangle r;
Code f;

point(b, p, f)          /* draw the pixel at p in b using Code f */
segment(b, p1, p2, f)/* draw the half-open[3] segment p1...p2 in b */
rect(b, r, f)           /* draw the half-open rectangle r in b */
bitblt(dst, p, src, r/*f)he ubiquitous bitmap copy operator */
texture(dst, r, src, f)* "paint" r in dst using the pattern in src */
```

Of course, `bitblt()` is the vogue term for rasterop. It copies a source `Bitmap` (given by `src, r`) into the destination `Bitmap`, `dst`, starting at `p`. Similar routines exist to draw splines, conics, boxes, shadows, text, and so on.

2.4. Windows

Windows are `Bitmaps`. In particular, a process using our routines has a global bitmap called `Display` which is its window on the screen. The full screen bitmap – called `Screen` – is also available. A program receives a default window and screen device, analogous to UNIX notions of standard input, standard output, and "/dev/tty". A program may operate on its window by supplying the `Display` bitmap as a parameter to a drawing function, or on the full screen (bypassing clipping) by using the `Screen` bitmap. For the programmer, no distinction is made between on-screen bitmaps (windows) and off-screen bitmaps. At a lower level, of course, such objects are handled differently, but for the application writer it is important that this abstraction be preserved. For example, to clear the window:

```
rect(Display, Display->r, F_CLR);
```

2.5. Input and Output Devices

User input is available either on demand (i.e. by polling) or asynchronously. Input devices must be requested by a program, or the window system will ignore any data generated by them. `InitDevices()` is used to request input devices. Supported devices include:

MOUSE	3-button optical mouse
KBD	keyboard
TOUCH	elographics touch screen

[2] Any Sun drawing code can be used, but in practice, these four are sufficient. Additional Sun drawing codes are useful for a few optimizations.

[3] Rectangles and segments are *half-open*; that is, they cover points up to but not including c. This simplifies tiling of rectangles and dot-to-dot line drawing.

There is a notion of *logical devices* – devices which are not directly supported by the system. A logical device may be anything for which there is a UNIX file descriptor; examples include files (like stdin), sockets, and special files.

When input arrives from some device(s), the system invokes a user-supplied routine. By default, a function named Input() is called, though the user is free to assign any routine to an input device. So for example, a simple program which draws "hello world" on a window each time input is present on the mouse or keyboard might be:

```
#include <sun.h>
Input(){
    string(Display, Dr.o, defont, "hello world", F_XOR);
}
main(){
    InitDisplay(); /* create the Display window */
    InitDevices(MOUSE|KBD);
    Go();
}
```

Go(), typically the last line of main(), "kicks off" the window system's input scanning[4]. string() draws a string in the specified bitmap, at the indicated point, in the supplied font; Dr is a common abbreviation for Display->r; and defont is a globally defined variable which contains the *default font.*

To obtain input, the Poll() routine must be called when input is present, for example,

```
devices_ready = Poll(MOUSE|KBD);
```

The mask returned by Poll indicates the devices with input ready for processing. State-oriented devices, such as mice, have a global data structure which may be interrogated as needed. For the mouse, this data structure is:

```
struct {
    short   buttons;  /* mask of buttons that are down */
    Point   p;        /* current location */
    short   own;      /* true if mouse in Display */
} mouse;
```

The state of the mouse buttons may be accessed through macros, e.g.

```
#define button1()  (mouse.buttons&BUT1)
#define button12() (mouse.buttons&(BUT1|BUT2))
```

So, to track the mouse while button 1 (the left button) is held the following could be used:

```
while (button1()) {
    ...
    Poll(MOUSE);
}
```

Touch screens and tablets are similar, but more like a one-button mouse.

Keyboard input is provided to the window system in an unencoded format. By default, the user receives keyboard strokes as ASCII characters, but a global state table permits the up-down

[4] More precisely, Go() puts the Display window on the Screen and begins an infinite select() loop, waiting for input or a timeout before calling Input;

state of individual keys to be examined. This allows *chording*. The routine kbdchar() is similar to the familiar getchar(). It returns the next character typed at the keyboard (with full read ahead), or -1 if input has been exhausted. So, to read a newline-terminated string from the keyboard:

```
while ((c = kbdchar()) != '\n'){
    ...
    Poll(KBD);
}
```

The treatment of logical input devices varies slightly from those devices the window system understands. To define such a device, one calls DefineDevice() with a file descriptor and (optionally) a routine to be invoked when input is available; e.g.:

```
extern InputFromNet();
NET = DefineDevice(netfd, InputFromNet);
```

If a routine is not supplied when a logical device is defined, the default Input() routine is used instead. The device identifier returned (NET) may be Polled like any other device.

Output readiness is handled similarly to input, though a bit less cleanly. A program may be notified when output is possible on a device, either by having a routine invoked, or by polling. A global variable, OutputReady, contains a bit-mask of devices to which output may be performed (as returned by select()).

For a last example, here's a simple program that draws a grubby finger print on the screen each time it is touched:

```
#include <sun.h>
#include <large/thumb>          /* three grubby finger prints */
#include <large/middle>
#include <large/index>

Bitmap *B[3] = {&thumb,&middle,&index};
#define RandomFingerprint B[rand()%3]

Input(){ /* when touched, draw a random fingerprint centered under touchpoint */
    Bitmap *b = RandomFingerprint;
    Poll(TOUCH);
    if (!touching) return;
    bitblt(Screen, sub(touch.s,Center(b->r)), b, b->r, F_OR);
    while (touching) Poll(TOUCH);
}

main(){
    SetDisplayRect(Rect(0,0,1,0)); /* a tiny Display window */
    InitDisplay();
    InitDevices(TOUCH);
    Go();
}
```








3. User Interface Tools

For building interfaces, we use libraries of graphics tools and some interactive editors. Libraries provide tools like *popups*, confirmation, text and keyboard handling, and frame-based text editing. There are editors for designing bitmaps (icons), fonts, and prototyping screen layouts.

3.1. Popups

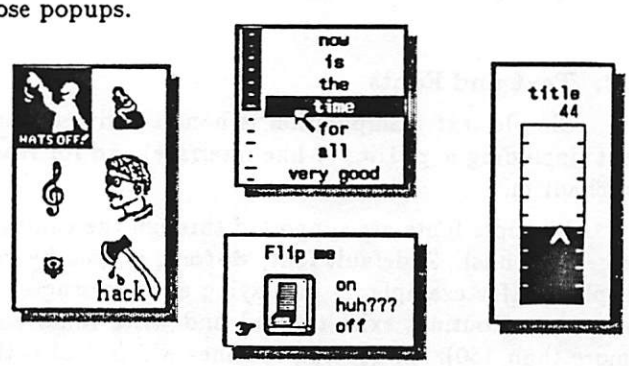
Popup menus and related objects such as sliders and multi-position switches may be placed on the display dynamically, in response to a user request, or statically (usually at the beginning of the program). This general class of interaction tools is loosely termed *popups*. When dynamically displayed on the screen, say in response to a button push, space covered by the popup item is automatically saved and restored.

Popups may be hierarchically organized to form a tree from which the user may make a selection. This organization allows for both "pull aside" placement and multi-level popups (a secondary item which doesn't pop up until it's parent's selection has been completed).

The function `Hit()` takes a pointer to a `Popup` and some acceptable set of push buttons, and returns an integer indicating the selection or `-1` if nothing was chosen. Functions may be associated with a popup, or any item in a popup. When a selection is made, the associated routine is invoked and `Hit` returns the function's return value. This facility has proven useful in creating more modular code for special purpose popups.

Several kinds of popups exist:

`Menu`	text menu
`BMenu`	bitmap menu
`Slider`	graphical potentiometer
`Switch`	2-3 position switch
`HMenu`	help menu
`FMenu`	text menu in multiple fonts



A text `Menu` displays all strings in the default font. Menus with many items may use a scrollbar to cut down on the number of items displayed at any one time. A bitmap menu is a rectangular array of bitmaps. A slider provides a numeric input mechanism with an internal integer range (the user visible range may be anything, provided the program supplies a suitable mapping function). Switches are either two or three position "throw switches". Help menus are used with bitmap menus to display the association between bitmap items and an appropriate text description. A font menu is simply a text menu in which each item may be in a different font.

The basic routines used with popups are

`New`*`Popup`*	create a new popup (e.g. `NewMenu()`)
`DrawPopup`	draw a popup on the display (used for static layout)
`UndrawPopup`	inverse of `DrawPopup`
`Hit`	handle user interaction with a popup

The system normally deals with keeping a popup's representation consistent with its contents. For example, if a menu's item is changed (say an item "open" is changed to "close") the next time the menu is `Hit` it will be correctly drawn.

As an example, here's a small program that creates a popup structure and presents it to the user in response to a mouse button being pressed.

```
#include <sun.h>
Menu *m
```

```
main(){
    Switch *sw; Slider *sl; Menu *ml;
    InitDisplay();
    InitDevices(MOUSE);
    s = NewSwitch("Wow! A switch!", "on", "off", 0),
    sl = NewSlider(5,5,25, "What do I do?", S_UP, 10, 0);
    ml = NewMenu("foo", "bar", "bletch", "burp", "<prev>" 0),
    m = NewMenu(
        "small",
        "big",
        "switch", M_SLIDERIGHT, sw,
        "slider", M_POPUP, sl
        "pull right", M_SLIDERIGHT, ml,
        "quit",
        0);
    Go();
}
Input(){
    Poll(MOUSE);
    if (button123())
        Hit(m, MouseButton);
}
```

3.2. Text and Fonts

Simple text manipulation is handled with a small set of routines which provide for drawing text (including a `printf()`-like interface) and for reading strings from the keyboard with a popup mechanism.

Multiple fonts are supported through the calling conventions to routines that deal with fonts (e.g. in menus). A default font, defont, is used by various library routines when a string must be displayed (for example, in displaying error messages). This font may be changed at any time by a program. Routines exist to read and write fonts and a sizable collection of fonts are available (more than 150). In general, routines which deal with strings and fonts are fairly naive. There is no direct support, for instance, for displaying strings in multiple inter-mixed fonts, this is left to the programmer.

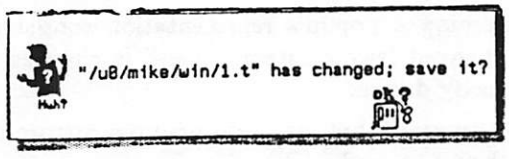
3.3. Confirmation and Messages

Mouse cursor confirmation is simple and useful:

```
if (Confirm(1)) { ... }
```

`Confirm(1)` waits for the user to press a mouse button and returns true if button 1 is pressed. A slightly fancier instance of this is the Message facility:

```
extern Bitmap *huh; char *filename;
if (Message(huh, 1, "%s has changed; save it?", filename)) {...}
```



This asks the user to confirm a write operation by pressing mouse button 1.

3.4. Text Frames

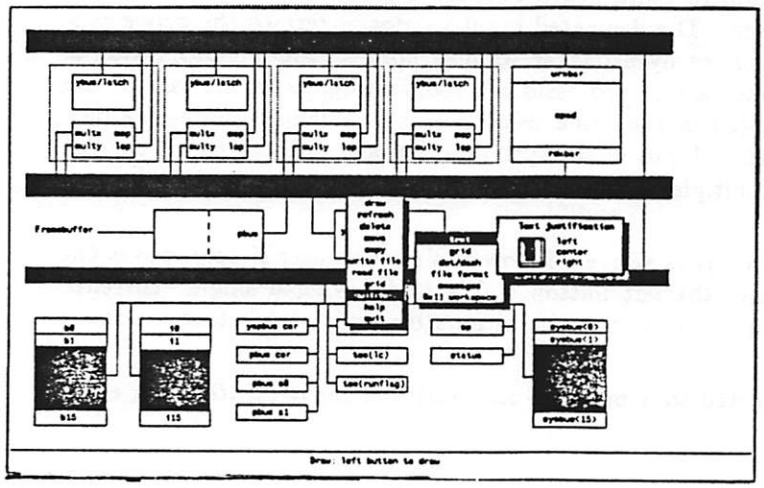
Text frames are a powerful mechanism for dealing with text input and display, rooted in the Smalltalk-80 [1], Blit, and Macintosh™ systems. A text frame is a rectangular region on the screen in which text may be displayed and edited with the mouse and/or keyboard. Text presented in a frame may be displayed in any font (one font per frame). Lines of text are either wrapped or truncated to fit a frame's dimensions. A scroll bar allows a user to peruse off-screen text in a frame.

Editing is cut/paste/copy style. Text selection may be made with mouse, touch screen or keyboard. More than a screenful of text may be selected by "dragging" past the top or bottom of the frame. Double-clicking selects a word, or a line, or a string of text enclosed by user-defined 1-character tokens. Menus may be used in conjunction with a frame to select a variety of commands such as regular expression searching.

Text frames have been used in constructing a number of programs ranging including a text editor, terminal emulator, mail reader, and news browser. Text frames are discussed further in [3].

3.5. Screen Layout

Programmers are responsible for the placement of bitmaps, menus, text frames, and so on. To simplify this task the *layout* program may be used. *Layout* is an interactive drawing tool along the lines of *cip*, *gremlin*, and MacDraw™ which allows a user to create drawings from any of the drawing primitives supported by the graphics library. Drawings may be saved either as C programs or PIC input files. *Layout* is hardly a complete user interface management system, but it is a simple and useful way to quickly prototype screen layouts.



4. Window Management

Window management routines are concerned mainly with handling the intricacies of the overlapped window environment (clipping and refreshing); in addition, they provide calls to perform the following operations:

- move a window horizontally or vertically on the screen,
- reshape a window's dimensions,
- open or close a window (change to/from iconic state),
- push or pull a window to the top or bottom of the screen
- select a window as the current input window.

Window management is broken up into code which resides in each application (about 750 lines of C code if all the supported functions are included) and code which resides in the UNIX kernel (approximately 2000 lines of heavily commented C code, about 9Kbytes of text). In general we have tried to locate policy-oriented code (e.g. keystroke encoding) in the application, with only the bare minimum placed in the kernel.

4.1. Overlapping and Drawing

The handling of window overlapping is mostly a carryover from the original Sun system. Each user program draws directly on the bitmapped display by having the bitmapped device mapped into its address space. A global overlapping database is maintained by the kernel. Applications are notified of changes to a window with a signal and are expected to then update their internal clipping state by querying the kernel. This model results in less than optimal screen management (in terms of repainting damaged areas) and, in some instances when the system is heavily loaded can cause major annoyances as windows are repeatedly repainted.

4.2. Window Hierarchy

Windows on the screen are organized in a tree structure with a window's location in the tree reflecting its location on the screen. That is, the topmost window on the screen appears as the topmost window in the clipping tree. This hierarchy is used in clipping calculations and, minimally, in input multiplexing.

Window inheritance is implemented through a "/dev/win" device, a character special device available to each process in a manner similar to "/dev/tty". The "/dev/win" device is used by a process to identify its parent window (for inserting a new window in the hierarchy or for taking over it's parent window).

4.3. Input Handling

Input is dedicated to a single window or multiplexed between a group of windows according to a locator device's location on the screen. The dedicated input window is termed the *active* window and distinguished from *inactive* windows by a darker window border. Input multiplexing is provided to handle the case where a collection of processes are cooperating, as in the case of the font editor, *fe*. Here each process is placed in the same *window group*. Window groups are patterned after the UNIX *process group* notion. Input in the system is actually directed to the current window group, which if it contains multiple windows, may require multiplexing based on a (mouse) device's location on the screen[5].

In normal interaction, setting the current active window is simple: a user simply moves the mouse to the desired window and presses the left button[6]. We find having a single "current" input window to be much less confusing to users than the alternative style of selecting the input window based on the mouse location.

At the lowest level, input is presented to a program as a series of *input events*. Each event record looks like:

```
typedef struct {
        short     ie_code;          /* input code */
        short     ie_flags;
#define IE_NEGEVENT  (0x01)        /* input code is negative */
        short     ie_locx, ie_locy;/* locator (usually a mouse) position */
        struct    timeval ie_time; /* time of event */
} InputEvent;
```

with the `ie_code` an indication of the type of input (mouse motion, keyboard, etc.). Keyboard input is presented to the application in an unencoded form. Locator devices have their coordinates translated to be window relative. While the input event mechanism provides full read-ahead on all devices, `poll()` routinely flushes old events on locator devices to avoid annoying "mouse-ahead".

A program must indicate it wishes to receive a specific class of input events for any to be

[5] If there is an ambiguity in directing input windows in a window group we resort to the window hierarchy.
[6] The window manager process also provides this command in its menu.

delivered (for example, all events associated with the mouse). This is normally carried out by the window system code and invisible to the programmer. If an input device is not enabled, events generated by the device are multiplexed according to the window hierarchy.

5. Design and Implementation Issues

5.1. Screen Refresh

An important design decision was that window overlapping should be transparent to applications. That is, programs should not be required to repair their window(s) when covered or uncovered. To do this most efficiently, we desire a `bitblt()` implementation along the lines of [4]. Unfortunately, given our constraint of using the Sun rasterop routines we were forced instead to use the Sun notion of a *retained pixrect* — an off-screen copy of the window. This incurs a significant performance penalty in normal operation: each drawing function (inside the Sun code) must draw twice, once into the clipped on-screen image and once into the off-screen copy. In certain instances, however, this can be a win. When drawing figures which are particularly expensive to clip to a window's boundaries (such as text), the items can be drawn into the off-screen image, then clipped once, as a unit, when placed on the screen (by *bitblt*-ing the affected portion of the off-screen copy onto the screen).

5.2. Subwindows

A basic choice in the window system design was not to support the Sun notion of a subwindow. In the Sun system a subwindow is a window which occupies a portion of another window's screen space and, in addition to providing a secondary clipping window, also contains a separate input/output stream and timeout mechanism. Subwindows are most useful (in our mind) in allowing multiple processes to share screen space without a priori knowledge[8].

A secondary clipping window may be obtained by creating a *sub-bitmap*, but the system does not directly support multiple i/o streams or timeout mechanisms. This means that it is up to the application for example, to multiplex several pseudo terminals, one per subwindow. Our decision not to support subwindows was based on the belief that the vast majority of interesting applications do not require this facility, and that supporting subwindows (in the Sun sense) directly in the window system would have greatly increased the overall complexity of the system. Our experience indicates this decision to have been reasonable.

5.3. UNIX Compatibility

We have tried to keep all window software compatible with accepted UNIX conventions. For instance, interactive graphics programs are free to use `stdin/stdout`, files, and networking facilities, and can be used in pipelines and shell scripts in traditional ways. Following are several examples.

5.3.1. Graphics Programs and Standard I/O

The command

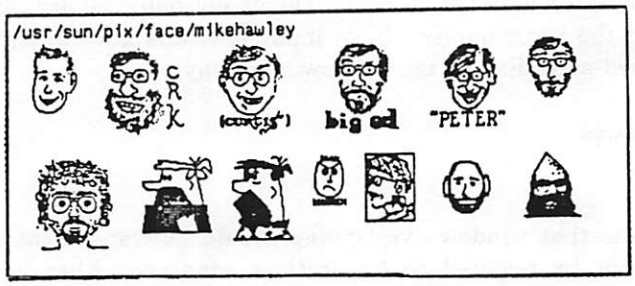
```
browse bitmap-files...
```

draws the `bitmap-files` in a window and prints the path name of each picture on the standard output when the user clicks on it with the mouse. *browse* is small — about 150 lines long. It can be used as a kind of *cat*(1) for bitmaps, and, since it writes selected names on the standard output, can be used like *pick* [6] in simple shell scripts when a picture choice is wanted, e.g.:

```
echo "Pick a face, any face..."
```

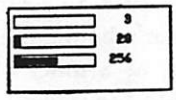
[8] Using subwindows to handle overlapping within a single process is also a normal reason cited for their existence; this we also believe provides minimal return given the added complexity.

```
face=`browse /usr/sun/pix/face/*`
...
```



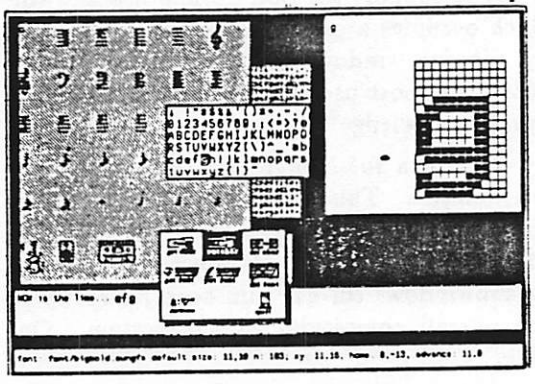
The *flow* command is a graphical *wc*(1). It draws a small *flow meter* window with three bouncing bar meters (displaying rates of characters, words, and lines). Because it reads from the standard input and writes to the standard output it can be used to monitor pipelines:

```
cat * | tbl | eqn | pic | flow | ditroff -ms
```



Aside from a small library of bar meters, *flow* is 75 lines of code.

A font editor, *fe*, required just a few days to implement, thanks to the fact that a powerful bitmap editor, *be*, was already available. Since a font is just an array of bitmaps, it was sufficient for the font editor to provide a small amount of font management (e.g., selection of fonts and characters) and connections to *be* for editing individual characters. The bitmap editor was modified slightly to accept commands from `stdin` in addition to the mouse. The font editor then simply opens a writable pipe to *be*, into which it sends a few commands (like, "clear the display, reshape the grid, and edit this file."). Not only was the font editor easily implemented, but the use of pipes gave the user a familiar editing interface — *be* — rather than requiring a special bit editor.



5.3.2. Design of Editors

The implementation of editors is another case in which the software design made possible by the single processor environment contrasts strongly with the Blit's two-processor world. On the Blit, an editor generally requires two processes, the main interface (in the terminal) and a supporting process on the host.

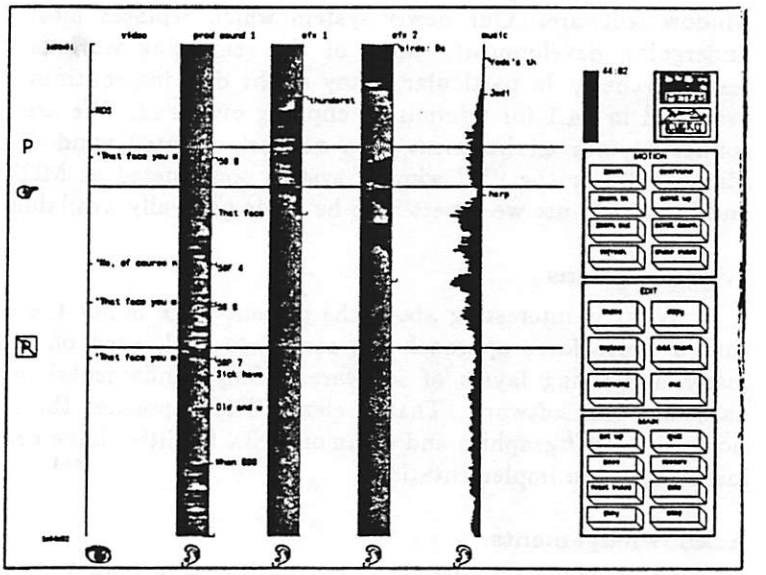
Some Blit editors, like the *icon* editor, require a fairly simple host part that interfaces the the interactive process in the terminal to the host file system in a straightforward way. Implementing a text editor like *jim* well is a more difficult matter [6]. Because the terminal has memory limitations, the management of files must be left to the host process, while the terminal maintains a current view of the files being edited. Since only the currently visible parts of the file can be stored in the terminal, operations like scrolling, cutting, pasting, and searching require substantial communication between host and terminal processes. It is necessary to optimize the representation of text buffers and maintain parallel data structures in both host and terminal components in order for the interface to present itself efficiently. The result is that roughly ¼ of the total code space (and substantial development time) must be devoted to the communication protocol, and considerable shrewdness spent on the problem of efficient memory management.

By contrast, we have implemented an editor called *fred* with comparable functionality and interface, and roughly the same binary size, but with a conceptually simpler underlying design.

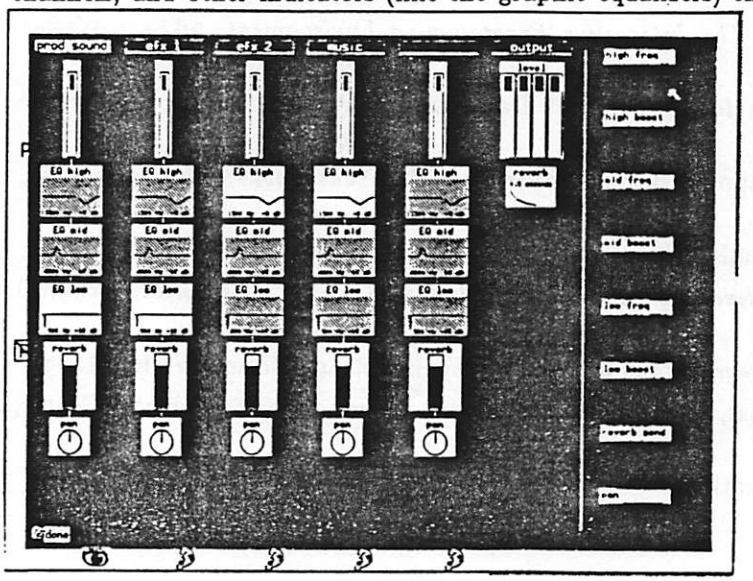
Since there is no communication constraint, all the routines for full text editing rest in one process. Although the *fred* editor is not as fast as it could be, it is as fast or faster than *jim*. Eliminating the slow communication link more than compensates for the less efficient design. Furthermore, the editing functions are available as a library, making it relatively easy to build custom interfaces that provide full text editing.

5.3.3. A Complex Interface for Film Sound Design

For a final example, here is an illustration of a complex interface for creating, editing, and mixing film sound tracks. The user presses graphical buttons on the touchscreen to edit film sound tracks and audition the material (in synch with video). At the touch of the *Meters* button, a screen full of meters and signal processing elements pops up.



Audio level meters (in logarithmic dB scale) bounce around in response to sound signals on several channels, and other indicators (like the graphic equalizers) change under user control. The main graphics interface runs on a Sun (possibly in conjunction with other windows) and communicates through a socket to a process on a different machine which monitors custom consoles and controls video hardware in real time. This process in turn communicates with a UNIX machine controlling the Lucasfilm Audio Signal Processor (ASP). Finally, there is a connection between the sound editor process and the ASP to provide for real-time display of sound.



5.4. Alternative Architectures

Older versions of our software which were compatible with the Sun window system required large binaries – starting with about 160Kbytes of low-level graphics overhead, the result of having to link rasterop and windowing libraries into every application. Huge binaries are inconvenient but not inherently evil. They run perfectly well, but burn disk space, utilize system resources, and tend to be awkward to compile and debug. Much of this code was unnecessary for us – like color support and high-level window management code, for instance – and was gradually discarded.

One obvious solution to this problem is to implement *shared libraries*. An alternative solution, one we prefer, is is to move most all the graphics code into a separate window manager process. This trades some potential responsiveness for vastly smaller binaries, as well as the advantage of full network access to the window manager. This approach, however, can require some rethinking in the construction of applications as the cost to access the screen goes up significantly.

6. Current and Future Work

The current system most people at Lucasfilm are using runs completely on top of the Sun window software. Our newer system which replaces most of the Sun windowing code is still undergoing development. Most of the remaining work is "tuning" the system for maximum responsiveness. In particular, many of the drawing routines do not take advantage of the scheme described in §4.1 for minimizing clipping overhead. We are also contemplating the possibility of connecting our environment to a network-oriented window manager, such as the ITC Window Manager [7] or the "X" window system constructed at MIT [8]. Once our new system is stable and in general use we expect it to be made generally available.

7. Conclusions

What is interesting about the present work is not the elegance of style so much as the fact that a brute force approach has served so well, even on a single-68010 4.2BSD system with so many underlying layers of software. Many fundamental optimizations could be made in both hardware and software. That is clear. The important thing is that clean and powerful abstractions like *bitblt* graphics and common UNIX facilities have combined in useful ways that outweigh inefficiencies in implementation.

Acknowledgements

We are heavily indebted to the work of Rob Pike, Luca Cardelli, and countless others.

References

[1] Adele Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley (1984)

[2] Michael Hawley, "UNIX Tools for A Personal Database," Portland USENIX Conference (1985)

[3] Rob Pike, "Graphics in Overlapping Bitmap Layers," *ACM Transactions on Graphics* **2**(2), pp. 135-160 (1983)

[4] Rob Pike, Leo Guibas, Dan Ingalls, *Bitmap Graphics*, SIGGRAPH '84 Course Notes (1984)

[5] Rob Pike, *The Blit: A Multiplexed Graphics Terminal*, AT&T Bell Laboratories Technical Journal **63**(8) (October 1984)

[6] Brian Kernighan, Rob Pike, *Programming in the UNIX Environment*, Addison-Wesley (1984)

[7] James Gosling, Dave Rosenthal, *ITC Window Manager*, Information Technology Center, (1984)

[8] Jim Gettys, personal communications.

A File System Tracing Package for Berkeley UNIX*†

Songnian Zhou, Hervé Da Costa, and Alan Jay Smith

Computer Science Division
EECS Department
University of California
Berkeley, California 94720

ABSTRACT

A tracing package for the UNIX file system has been implemented and statistics have been gathered from a heavily and widely used DEC VAX 11/780 running UNIX 4.2BSD. This tracing package is unusual in the comprehensiveness of the data gathered, the clean and usable format in which the final trace appears, and the use of a post processing step to assemble information in trace-records that is not easily (or at all) available at trace time. Trace records are gathered for file opens, file creates, file closes, reads and writes, renames, file deletes, executes, forks and exit calls. Some preliminary analyses of the trace data are presented. We found that the I/O activities are very bursty, that very few read and write operations are performed in most of the open-close sessions, and that the process lifetime distribution is highly skewed, with many short lived processes and a few long term ones. The extensive data gathered using the package is valuable for the studies of disk caching and file migration algorithms, distributed file system performance, and load balancing strategies.

1. INTRODUCTION

Over the past years, Berkeley UNIX has undergone extensive modifications and enhancements in its functionality and performance. In particular, the file system was largely reimplemented in 4.2BSD to provide features such as larger file blocks and contiguous block allocation [Mcku84], yielding considerably better performance. With the increasing complexity of the system, it has become more and more difficult to fully understand its behavior, and to identify performance deficiencies and bugs by reading the source code. A comprehensive trace-driven analysis of input/output activity can not only reveal many aspects of the system and user file access behavior, thus indicating ways to improve system performance, but also will provide invaluable data for the design of and research into disk caches, file migration, network file systems, and load balancing strategies.

In this paper, we describe a logical file system tracing package designed and implemented for the Berkeley UNIX system. We also present some preliminary results from analyses of the trace data generated using this package. Several goals and requirements for the package were set up to guide its design and implementation, and they are discussed below.

---

*The material presented here is based on research supported in part by the National Science Foundation under grant DCR-8202591 and by the Defense Advanced Research Projects Agency under contract N00039-82-C-0235. Partial support has also been provided by the Digital Equipment Corporation Eastern Research Laboratory.

†Unix is a trademark of Bell Laboratories

1) **Comprehensiveness.** The tracing package should provide a clear and detailed picture of the file system activities. For this purpose, we need to trace all the relevant file operations, and collect all the useful information. Efforts must be made to generate complete and accurate information in order to avoid as much guess work as possible in the analyses of the trace. The trace data should be useful for a wide range of analyses and simulation studies, many of which were unforeseen at the time the package was developed.

2) **Flexibility.** The package should be flexible enough to tailor to different tracing needs. Besides tracing the complete system, it should be possible, by specifying a number of parameters when activating the trace, to trace a single user, a group of users, a single process, such as a system daemon, or a group of processes. One should be able to activate and deactivate the trace dynamically without disruption of system services. The package should also be able to trace long term system behavior, as well as short term behavior.

3) **Minimum performance impact.** While a trace must by its nature generate some overhead, such a penalty should be kept to a minimum. When the trace is not activated, there should be almost no extra system overhead caused by the tracing package, and when activated, the tracing should not cause noticeable degradation to the system performance. There are two aspects to this: First, the extra amount of computing for extracting data from the system should be kept low. Second, the amount of trace data generated should not be so large as to constitute a significant portion of the system file activity. Both aspects are important if the trace is to run for an extended period, say several days, in order to eliminate the measurement error caused by temporary variations in system and user behavior.

4) **Minimum change to the system.** It is important to minimize the amount of code required for the tracer and to also minimize the degree of modification to the system. This not only makes the package easy to maintain and reduces the probability of the package introducing errors, but tends to reduce the performance impact on the system.

5) **Convenience for Analysis.** The trace data, in its final format, should be usable; i.e. it should be possible to analyze it easily and simply, with only simple computations required for simple outputs. Complex cross correlation of trace records should be done as part of the trace preparation, and should seldom be needed for trace analysis.

It is recognized that the above requirements often conflict with each other. However, our experience indicates that, by careful design and implementation, such conflict can be reduced and a reasonable compromise achieved. In the following sections, we will discuss how these requirements are met. The design of the package, including its general structure and trace data content, is described in Section Two. Section Three is a detailed discussion of the various issues involved in the implementation. We present some analyses results of the trace data in Section Four; potential uses of the package are also discussed there. The impact of the tracing package on system performance is quantitatively studied in Section Five. Section Six concludes this paper with an evaluation of the package.

2. DESIGN ISSUES

2.1 Design Considerations

The design of the tracing package follows the principle of achieving the functionalities required with minimum possible complexity. We first discuss the level at which the tracer runs and the reasoning behind our decision. This is followed by a general description of the package structure.

2.1.1 Level of tracing

Selecting the level of tracing is the first step. We want to trace the file operations performed by the users and system processes; this can be accomplished on top of the UNIX kernel. There are, however, several ways a user can perform a file operation: he can make a system call directly, or can use the system utility routines. This implies that the hooks for catching the operations must be placed in a number of places in order to capture all operations of the same type. Even so, there is still the risk of missing

some of them. Rather than performing the tracing at the user level, we decided to do it inside the kernel. UNIX has a small set of well defined system calls for handling all file operations, such as open, close, read, write, and rename. These calls provide a clean interface used by every file operation, thus enabling us to record all the operations by placing exactly one hook for each type of relevant system call.

There are two disadvantages to tracing at the system call level, however. First, by operating inside the kernel, the package has no way to tell where the file operation comes from. We remedy this by tracing the command that caused this operation, and matching the two records together later. Another problem is that the kernel has to be changed and debugging the kernel is generally much harder than debugging a user program; an error in the tracing package can potentially crash the system. We attempted to ease this problem by minimizing the changes to the kernel and by careful implementation. (This would have been easier had we been the only people attempting to modify the kernel at the same time.) Our results show that placing the tracer in the kernel is not a serious problem compared to the benefits achieved by an in-kernel tracing package.

Our interest in the trace data is primarily toward studying user behavior and related research issues such as disk caching, file migration, etc. Conversely, we are not strongly interested in "tuning" the system. Thus, our data collection is directed toward the collection of "logical" I/O information rather than "physical", and we want the logical read/write requests and not the physical block transfers to and from disk. Because of the complexities of tracing physical activity, such complexities including the use of a disk block cache and write behind, we have chosen to trace only at the system call level, ignoring some of the implementation details of the file system.

2.1.2 Structure of the tracing package

The package collects data about the file operations and records them in trace files. The package is composed of four parts. The first part provides the mechanism to activate and deactivate the tracing and to switch tracing files when they grow too large. Because of the wide range of system calls traced and the comprehensive set of information recorded, a large amount of data is generated in an actively used system; the limited amount of system disk space requires that we "ping-pong" between output files and dump them to tape as they become full. The second part of the package includes all the hooks placed in the relevant system call handling routines, and the corresponding routines to generate the trace records. The third part is the buffer management, synchronization, and file dumping routines. In order to reduce overhead, the individual records are not written to the trace file right after they are generated; rather, they are first accumulated in a large buffer in memory and then dumped. Since it is possible that several processes are trying to access a buffer at the same time, synchronization has to be provided to enforce strict time order of events and to avoid overwriting other trace records.

The last part of the trace package is not used during the generation of the trace, but rather at the *post processing* stage. Certain pieces of data are impossible to get during the trace, or are too expensive, in terms of processing time and additional data structures to be set up. Instead, we process the raw trace off-line by making several passes on it. The final trace generated from the raw trace and other information of the system is much easier to use by analysis programs. The details of the package implementation will be discussed in the next section.

2.2 Trace Data Content

2.2.1 What files are traced?

UNIX has a very general notion of files. Besides disk files, devices, such as terminals (*character special*) and disks (*block special*), are also treated as files and managed by the same naming mechanism. The inclusion of the IPC mechanism in 4.2BSD further extended the concept of files to include communication end points called *sockets*. Since we are primarily interested in the file system proper, all these special "files" are ignored by the trace package.

2.2.2 What activities are traced?

We trace the file **open**, file **close**, **read**, **write**, file **rename**, and file **delete** calls. In addition, three process control calls, **fork** (**vfork**), **exec**, and **exit** are also traced. Besides ordinary data files, there are also a few other types of files and file system internal data blocks stored on disk, including directories, symbolic links, and the file descriptors, called *inodes*. An inode contains information about a file, such as its owner, its size, its disk block addresses, and a unique number called *i-number*. In order to operate on a file, its inode must first be brought into the memory.

The files in UNIX are organized in a hierarchical structure. When a user makes an **open** call, s/he specifies the name of the file. Directories must be searched to find its inode. Because the way the directories are used is very different from ordinary data files and is not through the same set of system calls, tracing the directories would increase the complexity of the package by a substantial extent. Also, many directory accesses can be inferred, given full path names of nondirectory files accessed. Based on these two considerations, we decided not to trace directory activities. Similar considerations led to the exclusion of the I/O's for file inodes and indirect disk pointer blocks of files. Another type of special files is symbolic links. They simply contain names of other files so that multiple directory entries in different file systems can share the same file descriptor. We decided to ignore the symbolic links. Executable files in UNIX are accessed by the virtual memory system. The I/O operations on them are initiated internally by demand paging, rather than by explicit user requests. Although we trace the **exec** calls, which specify the executable files to be used, we do not trace the paging activities. (Paging activity can also, as an approximation, be inferred.)

In favor of simplicity, we ignored a number of I/O activities. This implies that a portion of the disk I/O is missing from the trace. Although this affects the accuracy of our data as a characterization of the system and user file operations, we feel that the most important and largest fraction of the I/O operations is still retained; much of the rest can be inferred with little loss in accuracy.

2.2.3 What information is recorded?

Table 1 is a list of the system calls traced and the data fields for each of them. Some of the data fields, such as the file close time in the open record, the complete file names, and the number of bytes transferred while a file is open, are impossible to derive during the tracing, at least at the proper time to insert them in the appropriate record. An important feature of the tracing package is the extensive amount of *post processing* performed on the raw trace to generate the richer and reformated final trace data. (A similar procedure was used to gather the data presented in [Smit85].) In the next section, we describe in detail the meaning of the data fields and the way they are derived, as well as the post processing.

3. IMPLEMENTATION ISSUES

We discussed the design of the tracing package in the last section, including the level of tracing, the package structure, and the data content. In this section, we consider a number of implementation issues in detail.

3.1 Synchronization and Buffer Management

Our scheme for buffer management is similar to the one used in an earlier file tracing package developed by Tibor Lukac [Luca83], and to the accounting mechanism available in 4.2BSD. As mentioned in the previous section, trace records are collected into a big buffer and dumped to the trace file in large blocks. When a system call of interest is invoked, the process enters a trace collection routine which allocates a section of memory from the trace buffer. If there is not enough space in the current buffer, a new buffer is requested from the system, while the old one is dumped and returned to the system buffer pool. Synchronization on the buffer must be provided, however, because the same buffer is shared by all the processes in the system. This turns out to be fairly easy because a process allocating space in the buffer is executing in the kernel mode, and therefore may only be interrupted by totally irrelevant system activities, such as the device driver. Since the space in the buffer is allocated in strict order of time,

RECORD TYPE	DATA FIELDS
open/create	*record type, record length, real time, process time, file id, open id, process id, user id, open mode, device number, last modify time, last access time, user name, file size when open, file size when close, file type, close time, number of I/O, bytes transferred while open, reference count, file name*
close	*record type, record length, real time, process time, file id, open id, process id, user id, open mode, device number, file size, file type, number of I/O, bytes transferred while open, reference count*
read/write	*record type, record length, real time, process time, duration, file id, open id, process id, user id, offset, bytes transferred*
rename	*record type, record length, real time, file id, process id, user id, device number, last modify time, last access time, user name, file size, old file type, new file type, old file name, new file name*
delete	*record type, record length, real time, file id, process id, user id, device number, last modify time, last access time, user name, file size, file type, file name*
execute	*record type, record length, real time, process time, file id, process id, user id, device number, user name, file size, file name*
fork	*record type, record length, real time, process time, vfork flag, parent process id, child process id, user id*
exit	*record type, record length, real time, process time, process id, user id*

Table 1. Trace Record Types and Their Data Fields (after post processing).

chronological ordering of event records is enforced.

3.2 File ID and Open-close Sessions

It is important for any meaningful analysis and understanding of the data that files have unique names. The path names of the files are not easy to use and are difficult to get during the trace. The i-number stored in a file's inode uniquely identifies the file within a file system; unfortunately, the i-number of a file is reused after the file is deleted, so it is only unique at a given time, and may not be unique for the duration of the trace. We decided to generate and assign unique *file ID's* to files during the trace. A file ID (identifier) is a number that is never assigned to more than one file for the entire duration of the trace. When a file is seen for the first time by an **open**, **rename** or **exec** call, we mark it by setting a flag in one of the unused locations in its inode. We also give it a file ID and record the ID in its inode so that next time this file is accessed, we can use its ID to identify it. These file ID's are just consecutively assigned integers, and function as unique, easy-to-use internal names of files.

Operations on a file generally follow the basic pattern of open/create - read/write - close. We call this cycle an *open-close session*. Tying together the data for a session provides insight into file access patterns; we use an *open ID* for this purpose. When a file is opened, an open ID is generated and stored in its inode so that all the operations on this file can be tagged with this ID. When the file is closed, the open ID is erased and never reused; next time this file is opened, a different open ID will be generated. These open ID's are also just consecutive assigned integers. There are two complications, though. First, the same file might be opened by several processes concurrently. Clearly, they should be considered as separate sessions. However, because of the constraint of available space in the inode, this scheme requires that they share the same open ID. In this case, the open ID alone is not sufficient to specify a session; the process ID supplied by the kernel must be used as well. Secondly, the same process might open the same file several times before closing it, in which case even the process ID is not sufficient. We choose to regard the multiple operations as belonging to the same session, so the pair {process id, open

id} can be used to provide a real unique session id. A field in the open record, the reference count, remembers how many opens have been performed on this file, thus giving an indication of the level of file sharing.

3.3 Fork, Exec and Exit

These three system calls do not describe file activity directly, but tracing them is necessary to correctly interpret the remaining data. For example, in UNIX, new processes are created by a **fork** or **vfork** call. All of the open files of the parent are inherited by the child, which means that it is possible for a process to open a file, do some I/O, and then fork a child process, which itself has a new process id. The child may do some more I/O and finally close the file. This sequence of file operations should be considered as a single open-close session, involving multiple processes. In order to recognize this sequence, the creation and destruction of processes must be recorded.

The **exec** call starts the execution of a new program. We would like to know which system command or user program initiated a particular open-close session, so that we can study the file I/O requirements and the CPU use of the various commands. This can be accomplished by tracing the **exec** calls.

Including these three calls in the trace greatly enhances its usefulness, since by looking at the trace, we know (almost) exactly what happened in the system. The CPU overhead caused by the generation of these call records is small and the amount of additional tracing code minor. For example, in a trace that we ran on a production system for a whole (working) weekday, the records of these three types constitute only 7.6% of the total number of trace records and 5.0% of the volume (bytes) of trace data generated.

The UNIX mechanisms of the inheritance of open files by child processes, the sharing of files by multiple processes, and the permissible multiple opens within one process make the recognition of an open-close session fairly complicated. By keeping track of process fork and exit calls, and by the use of open ID's, however, we have been able to identify all the sessions, thus making it possible to generate statistics such as the distributions of the number of I/O operations and the amount of data transferred in a session.

3.4 Complete File Names and File Types

We mentioned above that the user usually only provides partial file names; i.e. the user makes references relative to a working directory. Complete file names, however, are very useful; they often indicate the type and function of a file. For example, files in "/bin" are system provided utility programs, while files in "/tmp" are usually temporary files generated by system utilities such as the editors. File suffixes are also often informative. C language programs usually have the suffix ".c", while object files have ".o"; fortran files are identified by ".f". We have defined a number of file types based on file name prefixes and suffixes. File type information is generated during post processing and stored in the file open records of the final trace.

Complete file names are not easy to derive during the trace; instead, we decided to find them during the post processing. We record the parent directory information and the last component of the file name during the trace. Before and after the trace, we obtain a static picture of the file system. This information enables us to construct the complete names of most of the files by appending the last component to the name of the parent directory. The names of the files under directories that are created and destroyed during the tracing period cannot be constructed using this method. Since the directories are orders of magnitude more stable than data files, however, we do not believe we lose many file names. It would have been possible to construct all the complete path names had we traced the directory creation and deletion operations.

3.5 Times

There are two types of times in the trace records: the real time is the time since the trace started, and the process time is the total virtual time this process has consumed. All the times included in the trace are in milliseconds, but since the local VAX UNIX systems have a clock resolution of ten milliseconds, the times in the data collected are only approximate. The process time suffers from a larger

error margin because it is derived by a sampling process----which ever process is caught executing at the end of a ten millisecond interval is charged with ten milliseconds of CPU time, no matter how long it actually spent running during the interval. Despite this error margin, both types of times are very valuable for studying file operations and process behavior. For example, the real time durations of the user read and write calls are recorded, and can be used to evaluate the effectiveness of the read ahead and delayed write strategies in 4.2BSD file system. Lifetime distributions of processes, both real and virtual, are useful information for load balancing studies.

3.6 Post Processing

Post processing is an indispensable part of the tracing package. Although the raw trace contains (somewhere) almost all of the information that appears in the final trace, laborious processing would be required in any (and every) analysis to understand and interpret the data, since as explained earlier, the raw trace was generated with minimal overhead and consequently minimal user friendliness. We have chosen to write a post processing program which does that laborious processing once and for all. Post process consists of two phases: First, the raw trace is parsed to generate *session records* for each open-close session. A session record contains statistics such as the number of reads and writes performed during this session, and the number of bytes transferred. The close time and the size of the file at its close are also recorded. Static pictures of the complete file system taken before and after the trace are also processed to extract all the directories and their device numbers and i-numbers. In addition, a table is set up relating user ID's to user login names. Using the above pieces of information, the second phase of the post processing converts the raw trace into the finished format. Specifically, complete file names, file types, user names, and session statistics are all included in the finished trace data.

4. TRACE ANALYSES

In the above sections, we discussed the design and implementation of the tracing package in some detail. We are using the package to trace several machines in the EECS department at UC Berkeley, and in this section, we present some preliminary analysis results from one such (heavily used) machine; further data collection and analysis is continuing. An earlier study of this same machine appears in [Oust85], but relies on less complete and comprehensive trace data. Some of the measurements presented here include the distribution of the file I/O length and duration, the I/O transfer rate, the number of I/O operations within an open-close session, the process lifetime distribution and the distribution of the number of active processes.

4.1 General Statistics

The general statistics for the one-day tracing session are shown in Table 2. The machine was moderately to heavily loaded, with the load average (average number of ready processes) between 3 to 8, and sometimes as high as 12. The number of users ranged from 20 to 50. It should be pointed out that the data gathered are dependent on the system load characteristics, and will vary with the number of users and the type of work that they are doing; one cannot generalize from our data to different environments.

The distribution of the number of I/O operations per open-close session is highly skewed. For the files opened for read, 97% of them have less than 4 read operations. Eighty seven percent of files opened for write have only zero or one write operation. Roughly, two reads and one write are performed on the average during each session. The durations of open-close sessions are usually very short, on the order of one to a few hundred milliseconds. (70% of the sessions last less than 150 milliseconds, and only 17% of them last longer than 350 milliseconds.) The number of concurrently opened files is a few hundred under normal system load. The level of file sharing is fairly low. Of all the files opened, only about 8% are shared.

The observations given in the paragraph above illustrate a very important point: *the behavior of the system at a detailed and low level, such as that given by an I/O trace, is VERY hard to predict simply from a knowledge of what a typical user appears to do on the system.* It has been observed previously [Smit85] that the bulk of the I/O is generated by the system (only indirectly in response to user activity) and similar loose coupling between user behavior and system activity seems to also be present under Unix.

Machine: ucbarpa, VAX-11/780, 4.2BSD
Friday Apr 26 8:43 to 6:29 pm (9 hrs, 46 mns)
amount of data read: 385,646,844 bytes
amount of data written: 204,970,844 bytes
amount of trace data generated: 21,821,604 bytes
number of shared files: 7,977 (~8%)
maximum number of active processes: 189

Record Type	Count	Record Type	Count
number of records	538,588 (100%)	delete records	6,337 (1.2%)
create records	6,533 (1.2%)	rename records	168(3 in 10,000)
open records	87,664 (16.3%)	fork records	7,633 (1.4%)
close records	94,153 (17.5%)	vfork records	7,023 (1.3%)
read records	220,579 (41.0%)	exec records	11,641 (2.2%)
write records	82,266 (15.3%)	exit records	14,591(2.7%)

Table 2. General Statistics of a Tracing Session.

4.2 File I/O Transfer Sizes and Durations

4.2.1 File I/O transfer size and utility program behavior

Table 3 lists a few I/O sizes around which large numbers of the I/O operations concentrate; it is not a complete size distribution. Seventy-eight percent of reads and eighty-one percent of writes have sizes less than 40 bytes, and around 512 bytes, 1 KB, 4 KB, 6 KB and 8 KB. The percentage of reads with 512 bytes is unexpectedly high; so is the number of writes with 6 KB. Studies of the frequently used commands show that the text processing command "troff" reads source files in 512 byte blocks, and writes to output files in 6 KB blocks. We also noticed that the percentage of I/O's with 1 KB size is much higher than those with 4 KB and 8 KB. Since the 4.2BSD file system attempts to improve the performance by increasing the block size to 4 KB, and by doing read ahead, this indicates that many system utility programs are not optimized for the reimplemented file system. We examined a number of them by turning our tracer on, running the commands, turning the tracer off, and then looking at the trace output. Below are some of our observations.

- the text editor "vi" reads and writes in 1 KB blocks;
- the C language compiler "cc" reads in 1 KB blocks;
- the pattern searching program "grep" reads in 1 KB blocks;
- the remote copy (across machine boundaries) command "rcp" reads and writes in 1 KB blocks;
- the redirection mechanism (e.g., head file1 > file2) is not buffered, but rather sends data line by line.

We identified these block sizes without reading the usually complex utility programs, and the information is useful for the updating and tuning of system programs. (Some of these, such as the small block size for rcp, are not performance bugs; the ethernet board buffer is too small to accomodate much larger blocks [Cabr85].)

4.2.2 File I/O duration

Figure 1 shows the distributions of the durations of the read and write operations in real time. The file system performance improvements in 4.2BSD seem to be quite effective; most of the I/O operations return within 20 milliseconds.

I/O size range (bytes)	number of reads	percent	number of writes	percent
0-40	35,551	16.1%	18,191	22.1%
510-520	47,069	21.3%	1,076	1.3%
1020-1030	42,847	19.1%	20,946	25.5%
4090-4100	18,401	8.3%	3,332	4.1%
6140-6150	22,876	10.4%	22,979	27.9%
8190-8200	4,462	2.0%	1,756	2.1%
0-oo	220,579	100%	82,266	100%

Table 3. I/O Size Distribution.

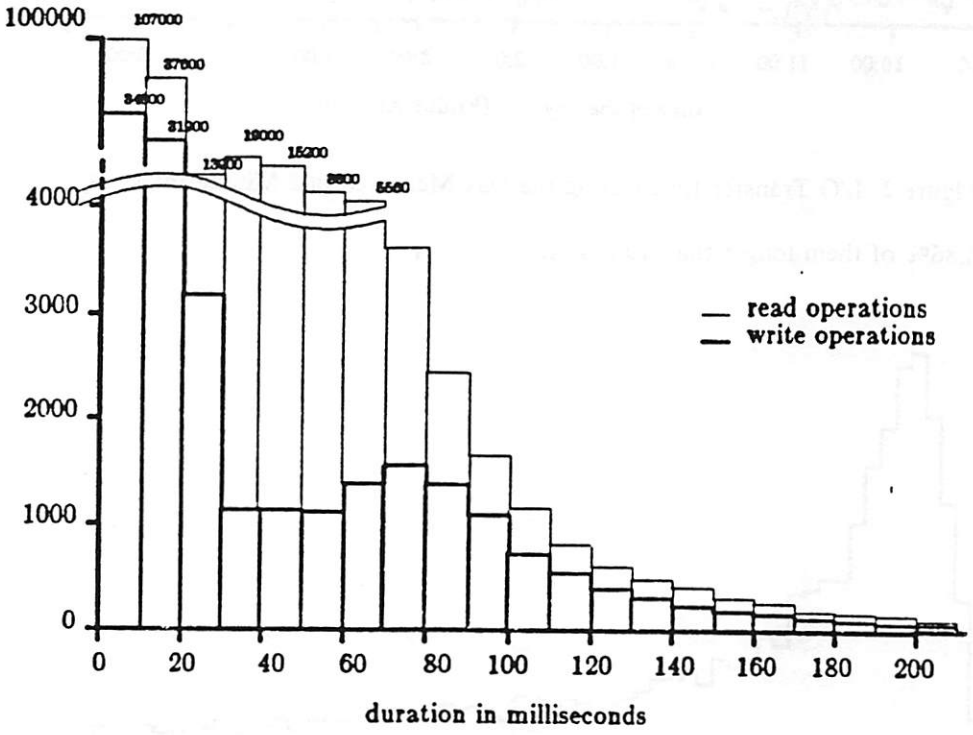


Figure 1. Distribution of File I/O Durations.

4.3 I/O Transfer Rates

The read and write transfer rate of the file system at a resolution of two minutes is plotted in Figure 2. We can see that the read and write rates are very much synchronized: when read activity is high, the write activity is high also. As might be expected, more data is read than written; the ratio of reads to writes is 2.68 and the ratio of bytes read to written is 1.88. We also observe that the I/O transfer pattern is quite bursty; the peak rate is around 3.7 megabytes/sec. for reads, and 3.3 megabytes/sec. for writes.

4.4 Process Lifetimes and Number of Active Processes

We observed earlier that the distribution of an open-close session duration is highly skewed. A similar pattern occurs, though less severe, in the distribution of process lifetimes measured in real time. Figure 3 shows the lifetime distribution of processes existing less than three seconds, which include 56.7% of all the processes observed during the entire trace. Thirty-seven percent of the processes last less than one second. Conversely, 26.7% of the processes exist longer than 10 seconds, 12.7% of them longer than

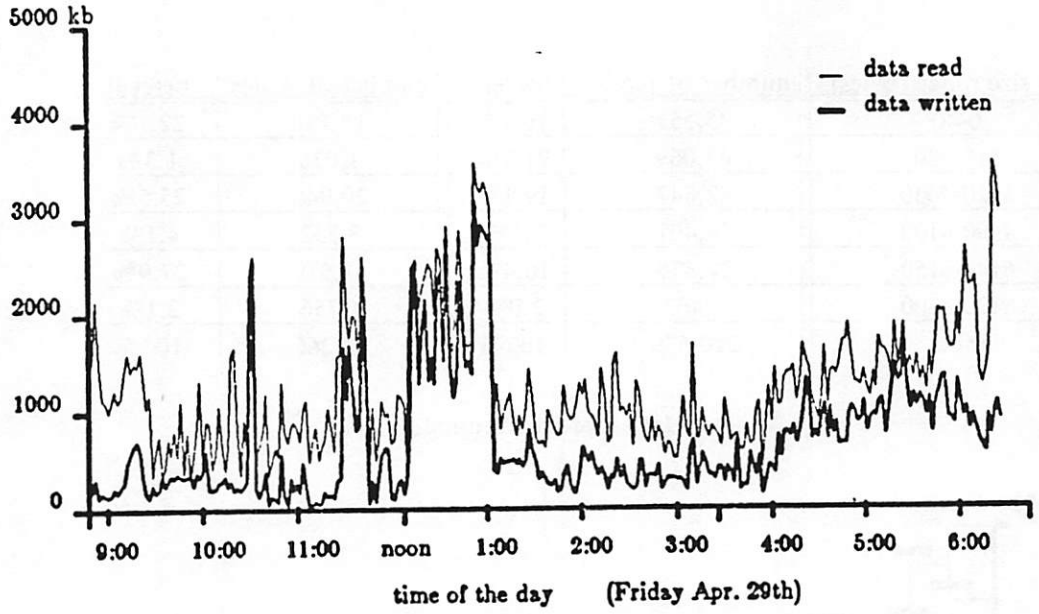


Figure 2. I/O Transfer Rate during the Day Measured in 2 Minute Intervals.

1 minute, and 2.86% of them longer than 10 minutes.

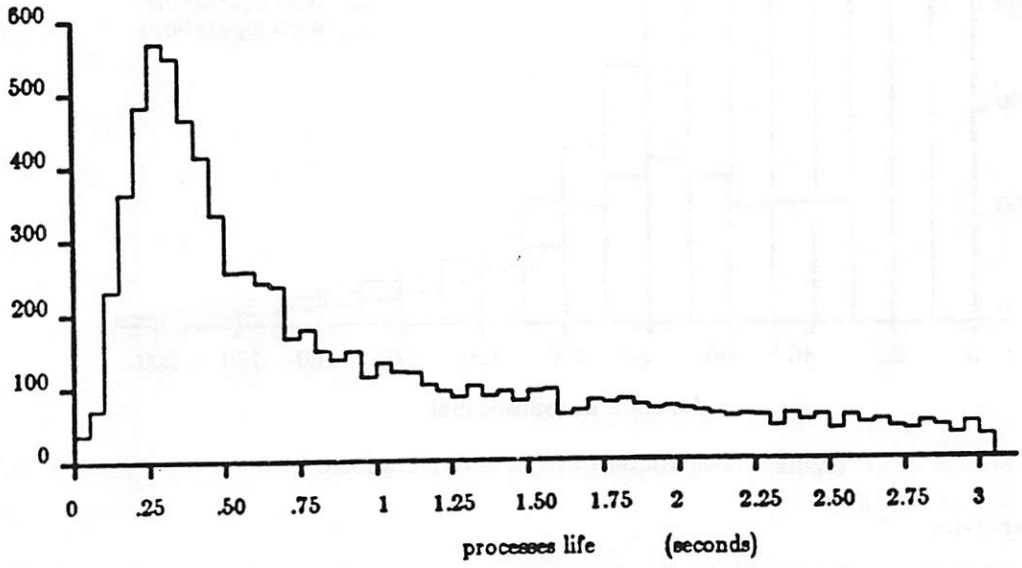


Figure 3. Process Life Time Distribution.

Although processes are born and die rather frequently, the number of *active* processes in the system (those that performed some file operations in an interval of a given length) does not fluctuate excessively. Figure 4 is a plot of active processes seen in one minute intervals throughout the whole day. We can regard this curve as an approximate pattern of system load. When the trace was started in the morning, mainly only secretaries were logged on, doing some clerical work; the peak load occurs around noon time, and then slowly decreases. (There is usually a pre-lunch peak, as everyone submits jobs before going to eat; there is often a pre-5pm peak as the clerical staff leaves, although it isn't evident in figure 4.) The

load remains high through the evening, as one might expect at a university, although it isn't shown in this figure. The maximum number of active processes observed is only 189.

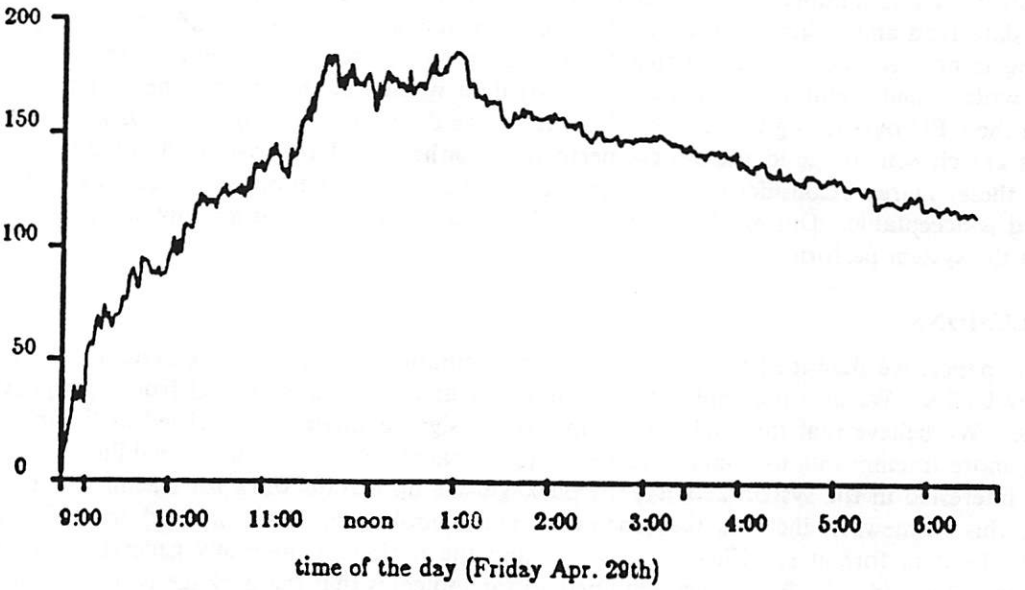


Figure 4. Number of Active Processes.

5. PERFORMANCE EVALUATION

In order to assess the impact of the tracing package on system performance, and to decide if it is suitable for extended tracing periods, we performed a few measurement experiments. Several factors contribute to the overhead incurred by the package. First, hooks are placed in relevant system calls and information gathering routines are executed every time these calls are made. Second, tracing information is written to trace files using the standard file system facility, thus competing with other activities in the system for I/O channel and disk bandwidth, and disk space. Finally, trace records are accumulated in large buffers before being dumped to the files, and new buffers are allocated on demand from the system buffer pool, hence adding to the contention for the buffers.

5.1 CPU Overhead

We wrote a test program that does nothing but a large number of file system calls in proportion to the distribution of the calls observed in the trace. The program was run on a VAX-11/750 a number of times with the trace turned on and off. There was no other user on the system; this way, we eliminated the error introduced by CPU and device resource contention, and by the fact that measured times under Unix are very sensitive to overall system load. Timing results of the program executions show that the test program consumes 7.7% more system time when running with the trace turned on than with the trace off. We can regard this increase as a measure of the overhead of the tracing on the file system calls. Since programs do other things besides making file system calls, it is reasonable to expect that the overall overhead of the tracing should be much lower than this number on a typical system. Although by running alone on a system, we avoided the error caused by interactions with other users, the resource contention overhead is also not measured in the above experiment. Hence, we performed the same set of experiments on the production machine, ucbarpa, on which the trace data were collected. The resulting overhead figures are slightly higher, but still below 10%. We also ran a few other I/O intensive test programs, and the relative increases in their execution times are similar to the numbers cited above. Another experiment with a CPU intensive program suggests a 2% to 4% increase in measured CPU time when the tracer is in use, although these numbers are very approximate due to load average differences and fluctuations between the trace/on and trace/off measurements.

5.2 I/O and Buffer Allocation Overhead

Besides the CPU overhead of the package, the above experiments also partly show the overhead on the I/O system and the buffer management. This overhead, however, can be directly assessed by looking at the amount of trace data written to the disk as a fraction of the total amount of data transferred by the I/O system. These numbers are included above in the general statistics of the tracing session. (The amount of data read and written in the statistics does not include the dumping of the trace data because the dumping is not recorded.) We see that the dumping constitutes about 3.6% of the total I/O calls (reads and writes), and about 9.6% of the volume of data written to the disks. These numbers are consistent with the CPU overhead given above. Since the trace data are written onto the disks in large blocks whose sizes are chosen for good file system performance, the actual I/O overhead should be somewhat lower than these figures. Considering the extensive amount of information we collect, we feel that such an overhead is acceptable. During the trace, we asked around and did not get any complaints from the users about the system performance.

6. CONCLUSIONS

In this paper, we described the design and implementation of a logical file system tracing package for Berkeley UNIX. We also presented some preliminary analysis results derived from data generated by the package. We believe that the package has met the design requirements specified in the introduction, except that more tracing options could have been incorporated to provide more flexibility. Although we are mainly interested in file system activity, the package sets up a framework for tracing any event inside the kernel; this framework includes the scheme to place hooks, the buffering and trace file switching mechanism, the data format specification method, and the notion of internally generated unique object ID's (file_id, open_id,...). Experience obtained so far indicates that the package is robust, easy to use, and relatively low in the overhead incurred. (For comparison, one of the authors of this paper found that tracing IBM systems using GTF caused CPU overhead upwards of 20%.) Since the size of the trace code is small, the tracer has the potential to be incorporated in the future as a feature of Berkeley UNIX. An embedded tracing package provides the system administrators the opportunity of turning the trace on whenever they want to examine the system operations and/or to detect performance problems, (or to collect data for research studies), without even bringing the system down in order to install a special instrumented version; the package can thus serve as a monitoring and performance debugging tool for the system.

The data generated by the tracing package can be used in a number of ways to provide insights into various aspects of the UNIX file system and user file access behavior. Although only some very simple analyses of the trace data are included in this paper, that data is informative. For instance, we found that I/O activity is very bursty, that very few read and write operations are performed in most of the open-close sessions, and that most processes are short lived. The analyses also exposed some unexpected behavior and some performance deficiencies in a number of frequently used system utilities, hence suggesting possible performance improvements. Besides studies of system behavior, the trace data can also be used in driving simulation models for the study of disk caching and file migration algorithms [Smit81a,b], and distributed file system performance [Porc82]. By providing detailed information about file and process activity, the data can also be useful for load balancing studies. Extensive analyses and simulation experiments on the trace data are being planned and will be reported in separate papers. We believe that the techniques used in the design and implementation of the tracing package are applicable in the construction of tracing packages for other systems as well, and that the results of the analyses will provide significant additional understanding of file system operations, above and beyond those aspects particular to the UNIX system.

7. ACKNOWLEDGEMENTS

The work presented herein would not have been conducted so smoothly without a number of people's support and help. We would like to thank Mike Karels, Kirk McKusick and Jim Bloom for their patience in explaining the UNIX kernel and system operation procedures, and for making available to us a number of the research machines for our experiments. Joel Emer and Joe Falcone of the Eastern

Research Laboratory of Digital Equipment Corporation provided some valuable suggestions during the development of the package. We are also indebted to many other friends for their help, including the plots in the paper.

8. REFERENCES

[Cabr85] Luis Cabrera, private communication.

[Luka83] Tibor Lukac, "A UNIX File System Logical Trace Package", Master Report, University of California, Berkeley, August 1983.

[Mcku84] Marshall McKusick, William Joy, Samuel Leffler and Robert Fabry, "A Fast File System for UNIX", ACM TOCS, 2, 3, August, 1984, pp. 181-197.

[Oust85] John K. Ousterhaut, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson, "A Trace-Driven Analysis of the UNIX 4.2BSD File System", submitted to the 10th Symposium on Operating Systems Principles.

[Porc82] Juan Porcar, "File Migration in Distributed Systems", Ph.D. dissertation, UC Berkeley, EECS Dept., June, 1982.

[Smit81a] Alan Jay Smith, "Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms", IEEETSE, SE-7, 4, July, 1981, pp. 403-417.

[Smit81b] Alan Jay Smith, "Long Term File Migration: Development and Evaluation of Algorithms", CACM, 24, 8, August, 1981, pp. 521-532.

[Smit85] Alan Jay Smith, "Disk Cache - Miss Ratio Analysis and Design Considerations", to appear, ACM Transactions on Computer Systems, 1985.

A Multiprocessor Performance Measurement Tool

P.K.Rowe
S.Sartzetakis
B.Vishnubhatla

ARTT Project
Dept. of Systems & Computer Engineering
Carleton University

ABSTRACT

The need for performance measurement of multiple microprocessor systems is discussed. A prototype tool which is used to measure the performance of a prototype system is presented. The test system is a real time embedded multimicroprocessor system, runing a multitasking multiprocessor operating system. The tool utilizes existing logic analysis equipment, a host computer, and a graphics terminal to provide both measurement and analysis functions. It provides a high level system designer's perspective and eliminates a need for low level understanding of the system.

1. Introduction

Multiple microprocessors have become a common element in current real-time systems design. They have distinct advantages in performance, reliability, and ease of design. However, these advantages bring increased complexities in several different areas. The first major problem is specifying systems with this level of complexity and concurrency. Another problem is partitioning the set of concurrent tasks onto different processors to achieve optimal performance. A further problem is evaluating the final product using the requirements specification, estimated performance from the task allocation, and the measured system performance.

It is generally accepted that current performance management techniques such as monitoring, tuning and/or upgrading the system, applied in an ad hoc manner, are not adequate, even for the classical case of time sharing systems. The central component of any performance management system is the performance monitoring system, which has not received the needed attention.

Even though system performance is predictable and a lot of work has been done in this direction with simulation and various modeling techniques [1],[2],[3], [4], only measurements can provide an accurate assessment of performance. However, extensive literature searches have shown that very little has been published on performance measurement.

Real-time systems, especially embedded systems, require extensive tool support. Performance monitoring tools are essential during the verification and maintenance phases, even though high performance systems can only be produced by considering performance factors during the systems' design.

This paper presents a prototype performance measurement tool that has been developed as a first step towards the evaluation of embedded multiprocessor systems. The principles, requirements, goals and main features of a general performance measurement tool of this kind are presented in the following section. The special target environment used is a prototype system

consisting of a multiprocessor system and an operating system combined for the first time for the purposes of the ARTT project. A description of the main features and characteristics of this environment, aimed at understanding the tool requirements, is given in the third section. The prototype tool is described next, and a simple application of its use is given.

2. General Case

In the general case, a tool of this type works for a distributed or a tightly coupled processing system. It includes a graphical representation of the concurrent tasks with direct output of the performance information on this graph. The performance information includes processor resource usage, messaging delay time, and latency measurements. Periodicity may be calculated or measured directly by exercising the system. These measurements are valid for systems with sporadic or deterministic measurements. Furthermore, both cases of data dependent and data independent processing are measurable and may be used for comparison purposes.

The first phase would be task histogramming. The percent of time each processor spends executing a particular task would be displayed relative to the other tasks. Histograms would show the user at a glance which tasks consume more processor resources and if usage is execessive, then modifications to the application may be in order.

3. The target environment

The target system for the multiprocessing performance measurement tool consists of a VME chassis, eight DY-4 DVME-102 single board computers, and the Harmony Operating System. Figure one illustrates the system in its environment. The DY-4 DSM 6816 shown is used for development since the Harmony Operating System does not support a development environment.

The hardware for the target system consists of a VME chassis, and eight MC68010 based processor cards. These DY-4 DVME-102 cards have the following features:

- MC68000/68010 processors running at 8/10 MHz
- optional memory management
- ROM monitors
- 256K of DRAM
- Serial I/O
- Timers
- Fully dual ported memory and I/O

This multiprocessor system has several specific hardware features to support the Harmony Operating System. First, the collection of processors is configured to have a linear address space. The memory map is shown in figure 2. This allows completely shared memory space using a unique address for each location. Second, interprocessor interrupts are provided using a write to a processor specific "magic" location. This allows each processor to interrupt every other processor in the system including itself. Third, by virtue of the use of the MC68010 processor, both test-and-set and multiple interrupt levels are provided. Finally, VME bus arbitration and access is provided for each card.

The Harmony Operating System is a real-time multitasking, multiprocessing operating system for embedded applications. It has the following feature set:

- send, receive, reply messaging primitives
- embedded computer support

- portability
- transparent multiprocessing
- multitasking
- extensive device support via servers
- real-time performance
- fixed priority scheduling
- hierarchical parent/child task relationships
- no task migration
- dynamic task creation

From a measurement tool point of view, the significant issues are the dynamic tasking, server support, operating system tasks, multiprocessing, lack of a development environment, and lack of task migration. Another target system measurement concern is VME bus saturation.

The server and operating system processes are important because they are viewed as part of the operating system. They constitute a part of the operating system overhead; however, they must also be considered as tasks in the system.

The multiprocessing issue is important because measurements must be made on different processors and between different processors. Since the proposed measurement is passive, hardware costs are high or distinct disadvantages are introduced.

The lack of a set of development tools as part of the run time environment has two influences. First, the system never incurs the development overhead; therefore, measurements are realistic. Second, the downloading of software requires time, and time delays such as this may be unacceptable.

The two issues of dynamic task creation and fixed task/processor bindings are important for several reasons. If tasks are dynamically created, it is difficult if not impossible to measure the data memory accesses of a task. Other methods must be used to estimate processor resource usage by a task. The lack of task migration simplifies the processor resource estimation for a collection of tasks of the same type because it is known that they always execute on one processor.

The last measurement issue is the critical estimate of VME bus utilization. Often, bus saturation is a major limitation for multiprocessing target systems with a single shared bus. The measurement or estimation of bus saturation is a critical parameter.

4. The current implementation

The goal of the current implementation was to provide a rapid prototype, which will form the basis for further developments. It serves as a 'system-breadboard' to try new ideas upon, and eliminate problems with the user interface. The objective is to build a simple tool to evaluate a measurement technique, and to aid in the incremental development of a general measurement tool.

The design of the prototype tool requires that the logic analyzer be sequencially connected to each of the target processors. The logic analyzer is remotely controlled by the host. It collects the information, and feeds the information back to the host. The analyzer is then moved to another processor and the same application process is executed. After all processors have been measured, a tool process on the host, analyzes the information and displays histograms of processor usage as a percent of the total time. These histograms are plotted on a Tektronix 4010 compatible graphics terminal, using the UNIX* functions tplot and graph.

* UNIX is a Trademark of Bell Telephone Laboratories, Inc.

4.1 Feature Set

The main features of the prototype tool were choosen to demonstrate the feasibility of such a tool and resolve issues associated with the user interaction. This features set is as follows :

- An estimate of the total resource usage for all tasks of one type is provided in a bar chart format.
- The tool is completely passive. It doesn't modify the execution sequence in any way, or change the load on the system.
- Estimates of the VME bus usage are included.
- Menu driven displays are provided.
- The graphical interface allows the display of selected tasks or groups of tasks by name, not address range.
- The tool provides storage of results and comparison with past results enhancing the system designer's view of system evolution.

The main issues for a system level, multiprocessor performance measurement tool were discussed in section three. A summary of how the prototype deals with these issues is given below.

(a) The servers, operating system tasks, and the operating system kernel, are part of the processor load. As such they must be included as part of the processing load and considered in the total load balancing. The tool supports these considerations.

(b) The multiprocessing measurements are made using a single Tektronix 1240 logic analyzer. An essential assumption in doing this is that the system is sporadic, and time averages are equal to statistical averages for the tasks in the system. This assumption reduces the cost of measurement considerably.

(c) The lack of a development environment in the target means that downloading the application is necessary. However, the current target downloading rate is 9600 BPS and this introduces significant delays. This problem may be overcome in the future by providing a switch between processors to eliminate downloading, or an Ethernet downloading link to eliminate the delay.

(d) The dynamic task creation feature of the Harmony Operating System means that the estimate of processor resource usage includes all tasks of one type. This is because instruction acesses are used to estimate processor resource usage. The fact that task migration is not alowed makes this measurement meaningful.

(e) Shared bus access is a limitation in many systems of this type. An estimate of the bus utilization is provided by summing the percentage VME bus acesses for each processor over all processors.

4.2 Hardware setup

The whole system can be viewed as a black box containing the Logic Analyzer (a Tektronix 1240 for total performance via a maximum of 72 channels), the program development environment (a DSM 6816 running UniPlus+ * UNIX System III), and their interfaces. Inputs to the system are the address, data, and control lines of the target processor(s), and the output is a histogram representation of the memory references of the different processors. Input data are taken through the input lines of the Logic Analyzer, and the output is directed to a 4010 type graphics terminal. There is a hardware imposed difficulty in the way the tool is presently implemented. There is no smooth way to switch the Logic Analyzer among the processors of the target environment.

* UniPlus+ is a Trademark of UniSoft Corporation of Berkeley.

4.3 Software package description

The software which is developed on the host, has two main functions. First, it automatically extracts information about the application tasks from the symbol tables, and constructs the setup files that are downloaded to the logic analyzer. This function integrates the logic analyzer with the tool, and saves the user from the need to program it through the front panel control keys, a complicated and error prone procedure. After the execution, it gets the data from the logic analyzer output files, and uses this data to plot the histogram graphs. The fundamental input to the graphics subsystem is the Logic Analyzer data files resulting from the measurements (see fig.3). The format of the output on the graphics terminal is an histogram representation of the memory references for the different processors. For this representation the use of 4010 termcap with tplot and graph utilities is being used. The tool stores the data for the graphics output in files on the host. In this way the user is able to recall and study older measurements, or even to compare them with new measurements by having the ability to display them on the screen simultaneously.

The tool is provided with a user friendly menu driven user interface which unifies the two basic parts of the tool, the data acquisition and logic analyzer setup on one hand, with the graphics generation on the other. It is designed to be modular, so it is easily extendable, and it is possible to switch parts, without discarding the whole system. For instance, the system and its backup files will still be useful if we decide to use a different graphics front end, and of course it will be useful for different kind of observations of the target environment.

4.4 Processor resource usage algorithms

The percentage of the idle time on the processor is directly measurable as the percentage of time the idle process executes. The current implementation uses a stop instruction in the idle loop to reduce memory contention; however, this makes idle time measurement more complex since the resource estimator uses memory instructions accesses, and the processor is frozen in the idle task. Unless further steps are taken, a distorted picture of the processor resource usage is presented.

The method used to improve the estimates uses a count of occurances of memory instruction fetches for each task in the system. The idle task is included in this count. Now, the idle task occurence value is increased using a multiplier. This multiplier is calculated using the mean time the idle task was idle, effectively adding the correct number of occurances to the idle process count. The sum of all counts gives total execution time and the percentage values for tasks can be directly calculated. The actual algorithm together with a terse description of each variable is given in the appendix.

5. An application

An application of the tool would be the following system. The target system contains two MC68010 based processor boards running the Harmony Operating System. The first processor runs five tasks : one application main task, and four of the operating system : __Directory, __Gossip, __Local__task__manager, and __Idle. The second processor runs three tasks : one application child task, and two of the operating system : __Local__task__manager and the __Idle.

In this simple application, main creates a task, child, which resides on another processor. The act of creating involves the __Local__task__manager task of each board. The __Directory and __Gossip tasks are for servers which are not included in the presented application and do not run.

The task main runs and alternates between :

(a). creating a child task ;

(b) executing a loop 10000 times ; and,

(c) sending ten messages to the child task.

The child task executes a loop a thousand times, receives a message, replies to the message and finally calls __Suicide.

Note however that message passing in the Harmony Operating System is a send__receive__reply construct. This means that if main sends to child, it must wait for child to reply -it is effectively removed from the ready-to-run queue and placed to a waiting queue. During this period, the __Idle task has the a chance to execute. The __Idle task has the lowest priority and only executes if nothing else is available.

Similarly for the second processor, if child tries to receive a message from main before one has been send, then it is placed on a waiting queue. During this interval, the __Idle task can run.

The percentage of time the __Idle task runs for each processor is an indication of the amount of unutilized processor time. If the percentage is high, then the processor is under utilized, and if too low, then processor utilization approaches the maximum of 100%.

6. Conclusions

The results from studying the prototype tool are summarized :

- The prototype system adequately illustrated problems in the design and use of a tool of this type to the extent that many problems will be eliminated in an augmented general tool.
- Approximately one hour of time is saved each time the tool is used to make a set of measurements which decreases the time required by 67%.
- The addition of an Ethernet board on the target is being considered. If the connection between the host and target was an Ethernet link rather than the currently used RS-232 link, downloading would be faster and considerable time could be saved.
- Measurement of the performance when there are dynamically created tasks, is not supported by the present prototype tool. The combination of the DARTT tool [5] and the general performance measurement concepts can provide this feature.
- The use of RAM packs in the logic analyzer is a major limitation and involves conciderable human interaction in an otherwise automated approach. The 1240 must store the performance analysis data in the RAM packs before transfer to the host.
- If the logic analysis probe could be connected to a box which has a switch and several dummy probes which can be connected to every processor, then the target software need only be downloaded once. After each execution, the switch can be moved to the next processor.

7. Acknowledgments

The work of the tool builders : D.Dodds, B.Jeddes, J.Nightingale, and A.Saikaley is gratefully acknowledged. The able help from Tektronix representetives A.St-Pierre, and D.Dypont was also appreciated.

References

[1]. Marsan,M.A.; Balbo,G.; Conte,G. ; Gregoretti,F. : 'Modeling Bus Contention and Memory Interference in a Multiprocessor System'. *IEEE Trans. on Comp.*, Vol c-32, No 1, Jan 1983, p.60-72.

[2]. Vrsalovic,D.; Siewiorek,D.P. : 'Performance analysis of multiprocessor based control systems [cm*]'. *Proc. of the Real Time systems Sympocium, IEEE* Arlington, VA., Dec.1983

[3]. Cho,C.K.; Lin,E.K.; Jen,C.L. : 'On Performance Evaluation of Multiprocessor systems for RT Simulaton'. *Proc. of the 17th Annual Simulation Symposium, IEEE* Tampa FL, March 1984, p.209-225.

[4]. Liu,J.W.S.; Liu,C.L.; Miyazaki,N.; Yamazaki,H. : 'Performance Evaluation of multiprocessor systems containing Special Purpose Processors'. *Podstawy Stereowania,* Tom 10(3), (1980), p.201-222.

[5]. Vishnubhatla, B : DARTT requirements and design specification. *Report ARTT 85-3,* ARTT Project, Dept. of Sys. & Comp. Eng., Carleton University, Ottawa, May 1985

PROTOTYPE TOOLS ENVIRONMENT

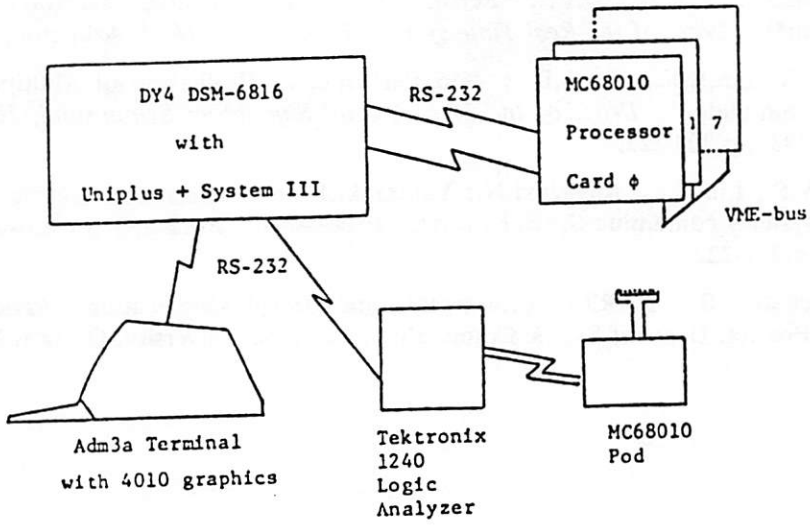


Figure 1

PROTOTYPE SOFTWARE ARCHITECTURE

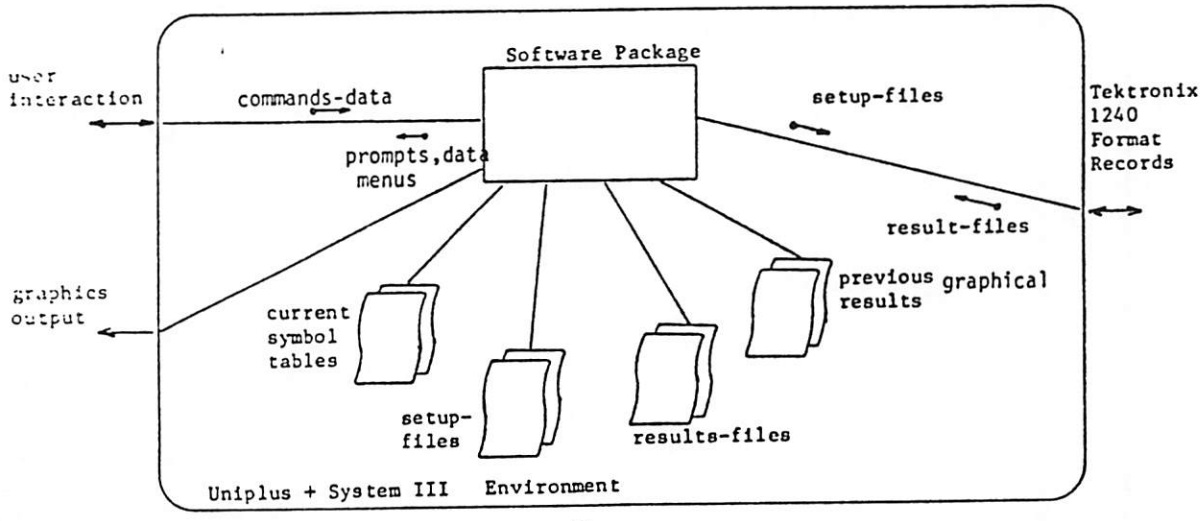


Figure 3

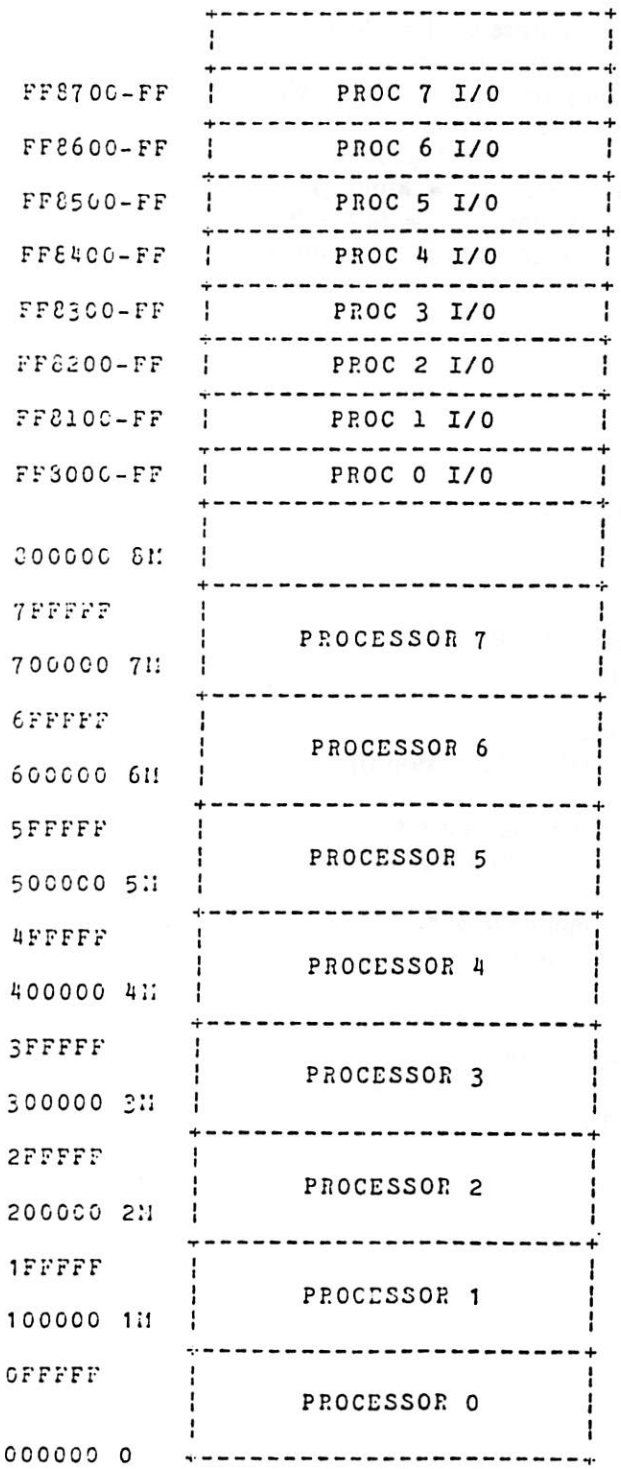


Figure 2:
VMEbus MEMORY MAP
FOR FULLY CONFIGURED
SYSTEM

Appendix

1. Algorithm Definitions and Pseudo Code Definitions:

process range count for process i on processor j = PrCnt(i,j)
total process instruction fetches on processor j = Tot(j)
maximum number of processes on processor j = Max(j)
percentage of processor resource of process i on
processor j = %(i,j)
multiplier for each task i on processor j = Mult(i,j)
mean value of idle process execution time = MVT(idle,j)
STOP instruction execution time on processor j = Tstop(j)

Pseudo Code:

```
for ( j=0; j!= max-processors; j++ )
   {
   Tot (j) = 0 ;  /* initialize total count*/

   for  ( i=1; i!=Max(j); i++ )
         {
           /* loop for each process value */
           /* first find multipliers */

       if ( i != idle ) Multi(i,j)=1
          else Multi(i,j)=MTV(idle,j) / (2*Tstop(j));

            /* compute expanded range values */
          PrCnt(i,j) = PrCnt(i,j) * Mult(i,j) ;

            /* now compute a running total */
          Tot(j) = Tot(j) + PrCnt(i,j)

          }
            /* end process computation */

   for ( i=1; i = Max(j); i++)

      %(i,j) = PrCnt(i,j) / Tot(j);
           /* compute percentage */

   }
     /* end of computation */
```

2. This is the code for the main task which runs on the first processor.

```
#include "/h/harmony/kernel-port/extern-and-.h/sys.h"
#include "/h/harmony/kernel-port/extern-and-.h/kernel.h"
#include "/h/harmony/kernel-port/extern-and-.h/templates.h"

extern main();
extern child();
extern _Directory();
extern _Gossip();
extern _Active ;
extern _Ready_;
extern _buserr;
extern _Excptv;

#define CHILD 20

unsigned _Pnumber = 0 ;

struct TASK_TEMPLATE _Template_list[] =
  {
         { 1, main, 250, 7, 0 },
         { 2, _Directory, 334, 7, 0 },
         { 3, _Gossip, 292, 7, 0 },
         { 0, 0, 0, 0, 0 }
  };

main()
  {
   unsigned child;
   int i;
   struct STD_RQST request;
   struct STD_RPLY reply;
   int dummy;
   unsigned outid;

   for (;;)
      {
          child = _Create( child );

        request.MSG_SIZE = sizeof( request );
          reply.MSG_SIZE = sizeof( reply );


        for (i=0 ; i<=10 ; i++)
        {

                  for (outid = 0 ; outid <= 10000 ; outid++)
                            dummy = i ;

                  dummy = i;
                  request.MSG_TYPE = i;

                  if ((outid = _Send( &request, &reply, child))!=0)
                  {
                            dummy = reply.RESULT ;
                  };
        };
      }
  }
```

3. This is the code for the child task running on the second processor.

```
#include "/h/harmony/kernel-port/extern-and-.h/sys.h"
#include "/h/harmony/kernel-port/extern-and-.h/kernel.h"
#include "/h/harmony/kernel-port/extern-and-.h/templates.h"

extern child();
extern _Active ;
extern _Ready_;
extern _buserr;
extern _Excptv;


#define CHILD 20

unsigned _Pnumber = 1;

struct TASK_TEMPLATE _Template_list[] =
  {
          {CHILD, child, 400, 8, 0 },
          { 0, 0, 0, 0, 0 }
  };

  child()
    {

      int i;
      struct STD_RQST request ;
      struct STD_RPLY reply;
      short int counter;
      int dummy;
      unsigned sendid;

      request.MSG_SIZE = sizeof(request);
      reply.MSG_SIZE = sizeof(reply);
      counter = 0;

    for (i=0 ; i<=10 ; i++)
       {
          for (dummy = 0 ; dummy <= 1000 ; dummy++)
                    counter = 0 ;

          dummy = i;
          sendid = _Receive( (char *)&request, 0 );
        reply.RESULT = counter++ ;
        _Reply( (char *)&reply, sendid);

       };

    _Suicide() ;

    };
```

Experiences with Electronic Software Distribution

Catherine A. Brooks

AT&T Information Systems
190 River Road
Summit, NJ 07901
attunix!cath

ABSTRACT

AT&T began electronic distribution of some of its software development tools in January, 1985. This delivery channel, called the UNIX™ System Toolchest, is one of only a few commercial ventures in electronic distribution and is the first of its kind for UNIX software. Through the Toolchest, programmers can browse, order, and receive source code electronically. The software is shipped under licensing terms that allow for unlimited copies of the "as is" software to be used internally.

This paper will discuss the Toolchest as an experiment in electronic software delivery and will outline some of the ideas, successes, and remaining challenges of using communications technology to distribute software.

The Toolchest is a straightforward application built with UNIX system tools. From a configuration viewpoint, two machines are used: a Browsing machine and a Distribution machine. The Browsing machine contains the catalog of tools and the customer records; it is here that customers log in. The Distribution machine, not accessible to customers, contains the warehouse of tools and distribution software. (See Figure 1.) The architecture of both the browsing and distribution components will be discussed below.

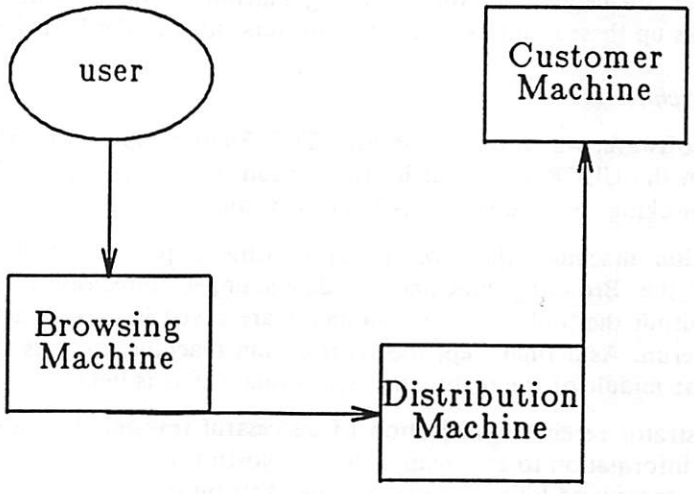


Figure 1. Toolchest Configuration

1. Browsing Architecture

The Browsing software was built using an AT&T-developed general purpose forms p ackage called *Tabs* [1, 2]. *Tabs*, written by Doug Blewett (now in Research at Bell Labs), melds qualities from both menu and block mode electronic forms into one unified interactive system that meets the Toolchest need for a flexible screen interface. With the *Tabs* collection of C subroutines, CRT windows are constructed with a portable virtual terminal library, and forms are constructed by calling *Tabs* procedures or primitives from some interpreted or compiled language, such as C.

Some of the *Tabs* features used in the customer interface software are:

- overlay editing.
- easy recovery from user-produced errors.
- help messages that are displayed in a separate "pop-up" window.
- scrollable regions (fields that may contain many different types of fields in a scrolling window).
- specification of field types.
- use of C functions for pre and post-processing the *Tabs* fields.
- *Tabs* control characters for users to request specific actions.

The browsing software was first executed on an AT&T 3B2 computer and then was p orted to a 3B5, as the number of dial-in ports was increased.

Anyone with a modem can dial into the Toolchest Browsing machine at 1200 baud, assuming they are using most CRTs in the UNIX system *terminfo* database. When they log in as *guest*, they will execute in a restricted environment that does not allow them access to the usual shell interface, but instead presents them with a menu.

Data on users, accounts, tools, news items, and purchases is stored in files of structures. Data in a user structure includes, among other things: an electronic mail address, a password, and the name of the machine on which to deliver software. A purchase structure includes details of the purchase, such as license fee, transmission fee, tool name, and billing information of the purchaser. An accounting structure includes details of each purchase, such as name, fees, tax, and date.

If a purchase (license contract, actually) is made, temporary files containing accounting and purchase structures are created on the Browsing machine. Nightly, the *cron* program runs a program that backs up these requests and then transfers them to the Distribution machine.

2. Distribution Architecture

The distribution software, which resides on an AT&T 3B20 computer separate from the browsing software, relies on the UUCP system at its foundation, thus relieving the Toolchest itself of the burden of error checking, communicat ions interfaces, and the like.

On the Distribution machine, the *cron* program runs a program that stores these requests transferred from the Browsing machine in date-stamped directories. It then creates *uuto* commands to transmit the tools. These commands are saved in case something goes wrong and they need to be rerun. As a final step, the Distribution machine executes the *uuto* commands to start delivery in the middle of the night when telephone traffic is light.

After the administrator receives notification of successful transmission, another program is run that sends billing information to an organization in North Carolina that generates paper bills. As with transmission, records of bills are saved on the Distribution machine.

Several factors led to the selection of the UUCP system as the delivery software of choice. First, UUCP is the only common communications package for AT&T UNIX system versions dating back to Version 7. The software has been widely ported to UNIX system derivatives as well, and successive updates to UUCP have carefully maintained upward compatibility, so we had the assurance that different versions of UUCP could communicate with each other.

Second, UUCP has the necessary communications capabilities. It permits communications over dial-up telephone lines, hardwired communications lines, and networks. It employs a robust packet, window-type protocol similar to the X.25 protocol that provides reliable data transmission. UUCP also has administrative software that allows for logging and status inform ation and forced retry. In short, it meets the Toolchest needs for an asynchronous UNIX system electron ic delivery mechanism.

Because some of the software in the Toolchest takes as much space as 750K bytes, mailing it to someone's electronic mailbox is unacceptable. The *uuto* command offers the advantage of making file and directory transfers to the *uucppublic* directory. So software and documentation are transmitted via *uuto*, and a *mail* message is sent to the receiver suggesting that the goods be picked out of *uucppublic*.

Herein lies the weak security point of the transfer. During the interval that the goods sit in *uucppublic*, the files have read/write/execute permissions by all. Because the Toolchest contract allows for multiple copies of source within a co mpany, copying is not the concern. The chance that the software could be tampered with while it waits in the *uucppublic* directory is a concern. We are in the process of implementing a scheme to lessen this possibility, and this effort is discussed in the section on *Challenges*.

3. Usage Patterns

During the first three and a half months of Toolchest use during a controlled introduction, data was collected on over 200 hours of connect time. Not surprisingly, the posting of the Toolchest phone number on the "net" on three separate occasions caused sharp increases in new users. We found that people logged in at all hours, with 8 to 9 am and noon to 5 pm being the peak hours, accounting for almost 60 percent of the sessions. The average length of a session was about 12 minutes, with most sessions lasting between two and 22 minutes. It also appeared that many people got timed out (when inactive for more than four minutes) and then logged in again.

On the distribution side, it has taken a while to hit our stride. Initially, every transmission required two to four attempts to complete delivery. One reason was that some time was spent tracing and working around a problem in previous versions of UUCP (see the section on *Adaptations*). Other problems resulted from incorrect UUCP login information being supplied on the forms from the customers, or from UUCP not being set up correctly on the receiving end. We also shipped the wrong version of the documentation once.

4. Successes

Several aspects of on-line browsing and distribution have worked particularly well. For example, the *Tabs*-based user interface for browsing has been well received, based on comments from users and the fact that it is used regularly without any documentation. Economically, electronic software distribution is proving itself to be attractive; the phone costs are recovered in the transmission fee and no one has to bother making and stocking tapes and paper documents. Another benefit has resulted from having a direct link to customers doing softw are development. We thus know who they are, how to reach them, and what they want in tools.

5. *Adaptations*

There were some surprises that already have resulted in adaptations. It wasn't long before we faced the possibility of sending out software packages that were nearly 4 megabytes long. Two things were done: the software was *pack*ed and then bundled with *cpio* to reduce its size and transmission time, and a ceiling was placed on transmission fees to reduce pressure to produce physical media.

In addition, we ran into a bug in early versions of UUCP that caused the first file of a transmission to fail with a "path denied" error because the directory did not exist on the receiving system. All later files, however, could be successfully transmitted. Our workaround was to include a dummy file as the first file in every transmission, thus resulting in harmless failures during transmissions to systems with older versions of UUCP.

6. *Challenges*

Many challenges remain. As mentioned earlier, we are implementing further security measures. The effort is based on the Automatic Software Distribution package [3] from Andrew Koenig in Research at AT&T Bell Laboratories. It is our intent to transmit a "sealed" package containing encrypted software and documentation plus a checksum of the plain text files. The package is then reasonably tamperproof while it sits in *uucppublic*. When the customer moves it to his or her own work area, it can be "unsealed" using the person's Toolchest password and an *unseal* program that will be sent to all Toolchest users granted licensing permission.

Other challenges include meeting the repeated request for an on-line demonstrat ion capability. This work in progress demands that security be given the most attention, thus ruling out many of the on-line demo capabilities that would work just fine if only someone could be watching over the user's shoulder.

Porting the software to take advantage of the sort of transparent file sharing over networks being developed for upcoming System V releases, is another effort that is underway so that many small Browsing machines may someday be able to share accounting files and a common customer database. This limits the impact of a single machine being down and it also accommodates the need for increased simultaneous access by maintaining good response times.

Adaptations for Europe and the Pacific are also in the works.

7. *Conclusion*

In summary, the Toolchest experiment has overcome the problem of keeping inventory of physical media by using the backbone of UNIX system intersystem communications to explore alternative means of software distribution.

References

[1] "Tabs Manual: Tools for Creating Window Based Electonic Forms," C. Douglas Blewett, December 7, 1983, TM 83-45411-9.

[2] "Tabs: A Window Based, Extensible, Highly Typed, Electronic Forms Package," December 7, 1983, TM 83-45411-10.

[3] "Automated Software Distribution," A. R. Koenig, USENIX Association 1984 Summer Conference Proceedings, Salt Lake City, Utah, pp. 312-322.

Documenting UNIX: Beyond Man Pages

Ariel Shattan
Jenny Hecker
Tektronix, Inc.
P.O. Box 1000, D.S. 61-261
Wilsonville, OR 97070

Traditionally, the UNIX* documentation set has presented problems for both new and experienced UNIX users. This paper discusses how the Tektronix ECS documentation group solved some of those problems.

When Tektronix' ECS business unit first started to build a UNIX workstation, the documentation group, newly formed and mostly UNIX-naive, was handed a set of Berkeley manuals and a set of AT&T manuals. We were given the task of learning UNIX from this documentation, and then writing a version of UNIX documentation to support our version of UNIX, UTek.

Marketing told us our typical users would be engineers or scientists, highly trained in their own fields. These people probably have used computers before, but they might not know anything about UNIX.

We were selling workstations rather than large computer systems, so, in many cases, there might only be one or two users who would have to set up and use the workstation without help from a "system manager" or field service representative.

Our UTek documentation should be designed and written so that workstation owners didn't even need to have a guru nearby, much less be a guru, in order to install, use, and administer the UTek system.

We needed to create a documentation set that would be useful not only to users unfamiliar with UNIX, but also to software engineers who have used UNIX for years. We wanted to make the documentation better without taking away the structure that old users are familiar with.

1. INTRODUCING USERS TO OUR SYSTEM

1.1 Reducing Intimidation

The traditional UNIX documentation set can be intimidating to a new user, with its huge, heavy volumes of technically-oriented tutorials and man pages.

The first thing our documentation group did to solve the intimidation problem was cosmetic: we made the manual set look smaller and less intimidating by going to half-size (6 x 9) manuals. This is nothing new in the UNIX world, but generally when the manuals are cut to half-size, the number of manuals increases proportionately. We managed to fit the UTek documentation into a relatively small set of books.

Still, there are enough books in the set to make new users think that the system must be hard to use if it requires so much documentation. To reassure them, we added the Documentation Map.

---

* Trademark of AT&T Bell Laboratories

1.2 The Documentation Map

The Doc Map (Figure 1) is the first thing users see when they open up the manual set. It shows the users where to look for information depending on: 1) what they need to know, and 2) what they already know.

What we try to get across early, through the Doc Map, the book titles and the introductions to each book, is that not every user needs to read every book in order to use the system.

If you're new to UNIX, then you have to begin at the beginning, but how far you go through the manual set depends on how much you want or need to know. If you're an old hand at UNIX, you can skip most of the basic information and use just the in-depth reference material.

1.3 Teaching UNIX

Almost everyone agrees that the best way to learn UNIX is to ask the local guru. But our workstations are designed to be run and administered by a single person or a small group of people, who may be UNIX-naive. There might not be a UNIX guru around to ask, so our user has to depend on the documentation to get started and to become proficient enough to do useful work.

Marketing had told us that most of our users would be college-educated. We had to keep in mind when designing the UNIX novices' information; we didn't want to insult the user's intelligence -- but we couldn't assume too much system knowledge, either.

1.3.1 How To Teach? We decided to make the user's first introduction to the workstation an interactive, online tutorial that lasts about an hour and a half. This tutorial is intended to help the user feel comfortable using the workstation, and to teach the essentials of UTek. We wanted the user to be able to do productive work after going through the tutorial. (However, the work the user can do after the tutorial probably is not totally efficient, since it's impossible to teach all the shortcuts in 1 1/2 hours!)

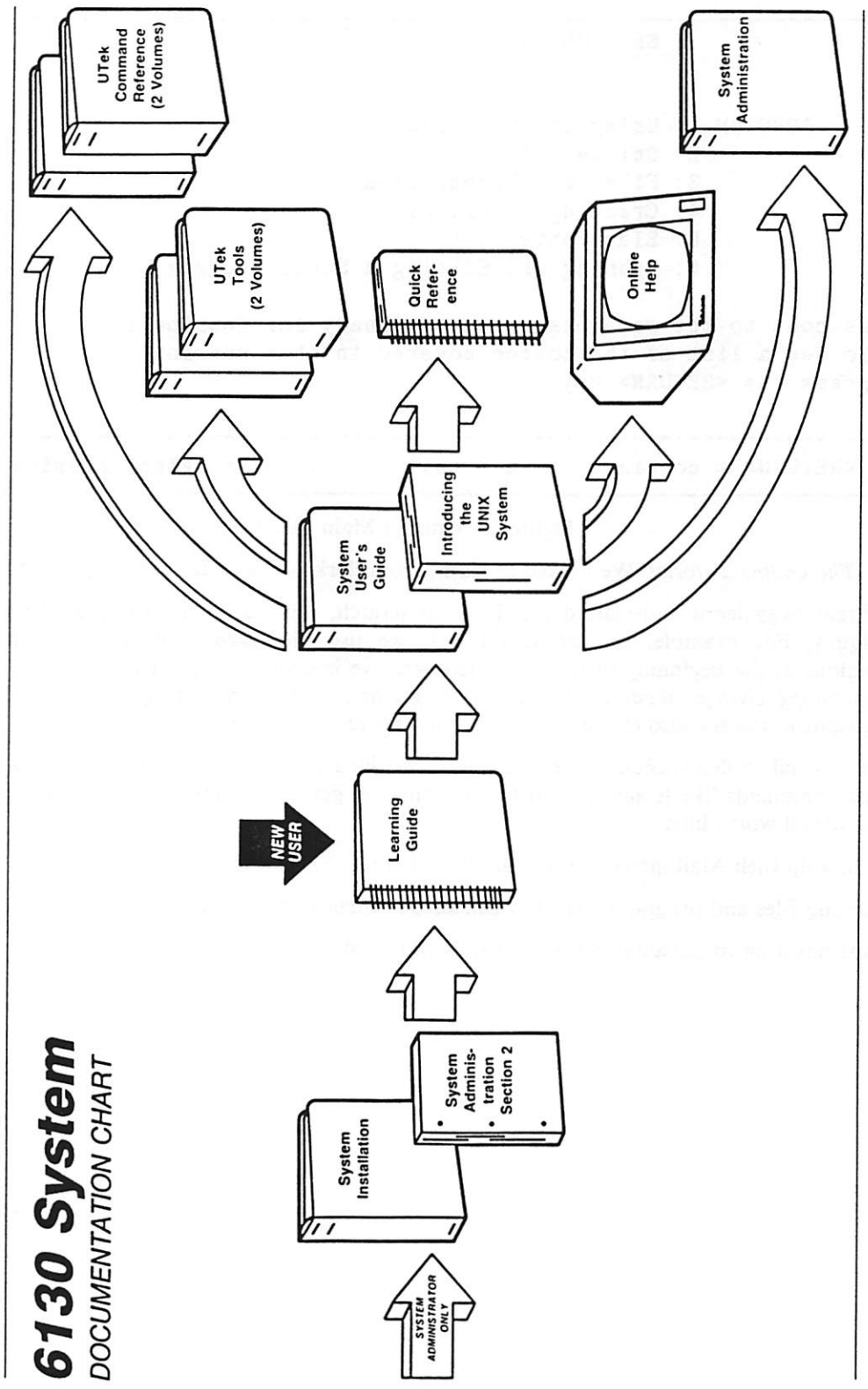


Figure 1. The Doc Map

This tutorial is not only useful to a UNIX-naive user. It can also teach a UNIX-knowledgeable user something about our workstation, since it explains areas like our mail system (a version of Rand MH mail) and help system. Figure 2 shows the main menu of the tutorial.

```
                SESSION MENU:


----> SESSION 1: Using the Sessions
              2: Online Help
              3: Files and Directories
              4: Creating a Text File
              5: Electronic Mail
              6: Running and Editing a Shell Program

  Welcome to the sessions!  You are ready for Session 1.
  To see a list of the topics covered in that session,
  press the <RETURN> key.

  ------------------------------------------------------------
   <RETURN> = continue      q = quit       1 - 6 = select session
```

Figure 2. Tutorial Main Menu

1.3.2 The Online Tutorial We looked at **learn**, from Berkeley, and decided we could do better.

We threw away **learn** and started mostly from scratch, but we do use some of the best **learn** techniques. For example, in our **vi** tutorial, we use the **learn** method of placing editing instructions at the beginning of the file. However, we improve the tutorial by re-using the same file, *including changes made by the user during edits*, all the way through the vi tutorial. The instructions in the file also change each time the user requests the file.

In our tutorial, called **sessions**, users are always working in their home directories. That way, we can use commands like **ls** and **pwd** in the sessions and get real results. During the sessions, users can do useful work, like:

- setting up their Mail inbox so they can use it later.
- creating files and programs that they can save in their home directory.

Sessions has a lot of advantages over **learn**, as shown in Table 1:

Action	in Sessions	in Learn
knowing where you are in tutorial	menu/page structure tells you where you are (Figure 3)	lesson number given, but no idea how many lessons there are
choosing lessons	can start with any lesson, skip any lesson through the 2-level menu	can start at a lesson number. No idea what's ahead/behind
moving around	can get back to menu, back up, or quit at any time. Choices always displayed on screen.	can back up or quit any time, but you must remember the command to type (like "bye")
reviewing	can back up 1-5 screens (to beginning of lesson). Can redo any lesson	can back up one page
	one-screen quick review at end of each lesson for reinforcement	occasional reviews
skipping questions	press ⟨Return⟩ key	sometimes possible, but you usually still must type "Ready" and then "n" when asked to redo
answering questions	2 chances; then correct answer given (Figure 3)	as many chances as you want. Never gives correct answer
restarting	2 levels of menus. You can recognize where you stopped	Must write on a scrap of paper the name of the next uncompleted lesson
accompanying documents	short booklet . Tells how to start. Has one page for each session: estimated time, subjects covered, quick review of most important info.	man page -- tells how to start and lists general subjects. Also a paper on how **learn** was created and how to make scripts for it

Table 1. Sessions vs. Learn

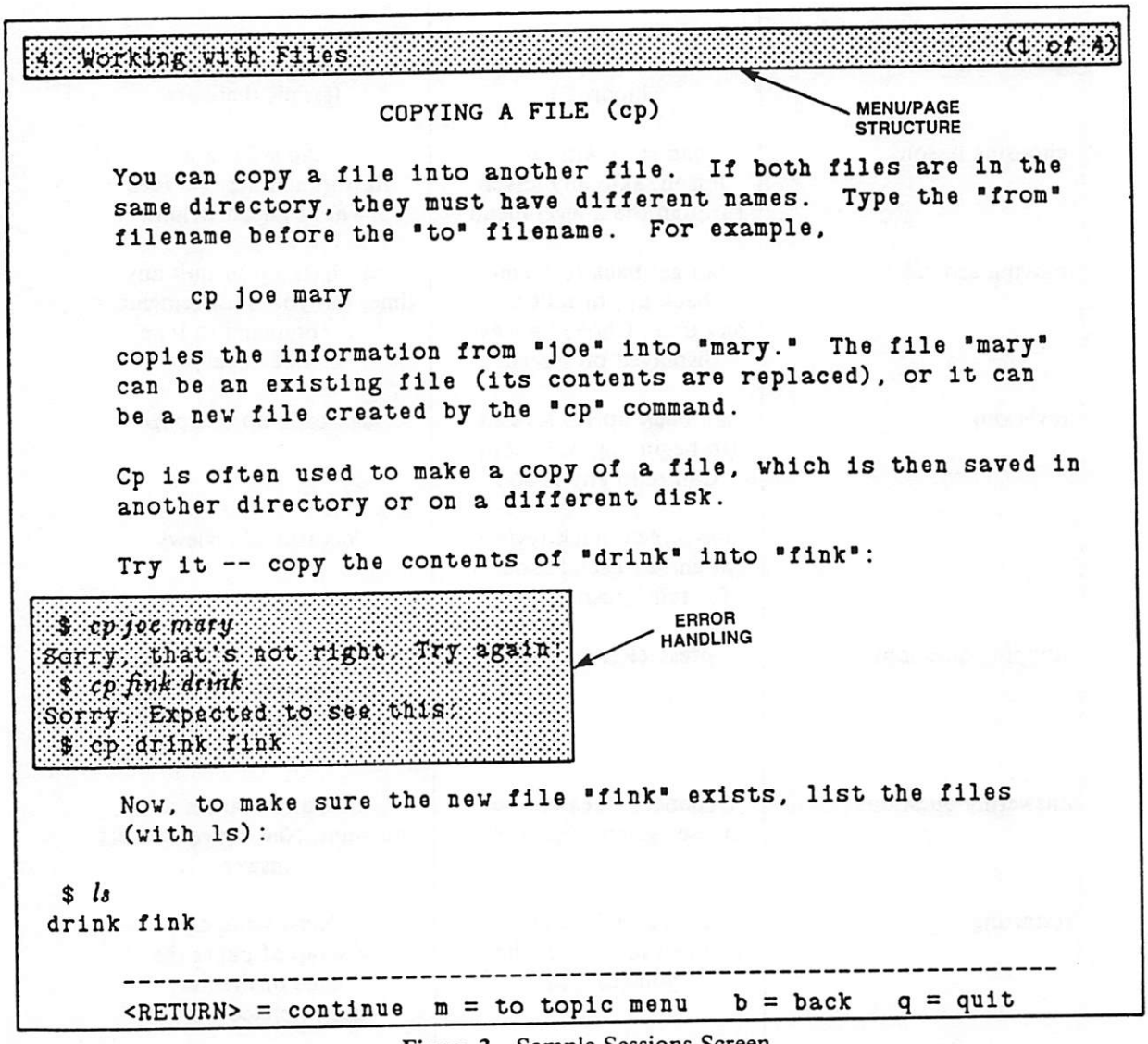


Figure 3. Sample Sessions Screen

If the users want to know more about UNIX after getting their feet wet with the tutorial, we direct them to our Introduction to UNIX book.

1.3.3 Re-Inventing the Wheel There are probably about 15 good "Introduction to UNIX"-type books on the market now. With all of those to choose from, we saw no reason to write our own, so we evaluated all the books we could find and chose the one that we thought best suited our audience -- it was clear and complete, and it had a tone that our audience wouldn't find condescending. The book we selected was *Introducing the UNIX System*, by Henry McGilton and Rachel Morgan. We distribute this book as part of our documentation set.

We include a cover letter that summarizes the differences between UTek and the UNIX described in the book.

2. ONLINE HELP

One of the things the online tutorial teaches is how to use the various kinds of online help we have on our workstations. How you use the help varies, depending on your level of UNIX experience.

2.1 For New Users

2.1.1 Online Man Pages On systems with enough disk space, we provide the full formatted manual pages (accessed with the **man** command, as usual). These man pages are discussed later in the paper.

2.1.2 Message Help There are two levels of messages on our system. The short, somewhat cryptic message you've come to expect from UNIX is still there. This message works well for users who know what's going on. We also provide extra help on most system and utility error messages for the new user.

Each short message has a *tag* at the end. By typing **msghlp** and the tag, you can get a little more information about the problem and its possible solutions (Figure 4).

```
$ more fakefile
more : fakefile : No such file or directory (sys2)
$
$ msghlp sys2

sys2:
"No such file or directory"
The file does not exist. These are the possible problems: 1)misspelling
the filename, 2) some component of the path doesn't exist, 3) using the wrong
path, 4) being in the wrong directory, or 5) not having the right permissions
to access the file. Try using the "ls -l" or "find" commands. (ENOENT)
```

Figure 4. Msghlp

2.1.3 The Help System There are two levels to this, too.

You can get online help on a specific command. This (discussed later in this paper) is most useful if you know what command you want.

Frequently, new users know what action they want to take, but not *how* to do it. For example, you might want to know how to check on a process running in the background. You can type **help**, and our Topic Help menu (Figure 5) appears. You can be reasonably sure that the information you need is under 1a Background Mode, so you select that option. The help screen in Figure 6 then displays.

```
                    Help System Topics

1. Command Entry and Execution
     a) Background mode
     b) Control characters
     c) Entering commands
     d) Pipes
     e) Redirecting input and output
     f) Parameter substitution
2. Files
     a) File operations
     b) Filenames
     c) Protecting your files
     d) Links
     e) Wildcard characters

3. Directories
     a) Directory operations
     b) Pathnames
     c) System directories

4. Other
     a) Archiving
     b) Environment variables
     c) Regular expressions
     d) Quoting special characters
     e) Status
     f) Local area network

Q. Quit

-----------------------------------------------------
Choose topic by number and letter, such as 1a:
```

Figure 5. Topic Help Menu

```
RUNNING COMMANDS IN THE BACKGROUND                    Screen 1 of 2

        If you want to keep a command from tying up your terminal while
        it is running, you can run it in the background. To run a
        command in the background, put an & at the end of the command
        line.

        Example:        nroff -ms memo > memo.d &
        Response:       2204

        The system responds with a number that identifies the background
        process.

        You can check the status of a background process with the ps
        command. The following example finds the status of the process
        in the previous example.

        Example:        ps 2204
        Response:       PID   TT  STAT    TIME    COMMAND
                        2204  01  S       0:10    nroff


-------------------------------------------------------------
Back    Main menu       Quit
Press N to go to next screen or type first letter of option:
```

Figure 6. Typical Topic Help Screen

2.2 For Experienced Users

All this help is very useful for teaching new users to use and feel comfortable with UNIX, but what are we doing for the folks who already know UNIX?

The traditional UNIX documentation consists of man pages and a collection of tutorial papers. We did not ignore this tradition when we created our document set. We provide man pages in the *UTek Command Reference* manual and online form, and we provide tutorials in the *UTek Tools* manual.

2.2.1 Man Pages Even the most experienced UNIX user has to look up information in the man pages, and some people have problems finding that information, whether they are referring to online man pages or paper copies. These problems stem not only from the highly technical tone of many man pages, but also from the fact that there is so much information in any given man page that it's difficult to find just the information you're looking for.

To combat these problems, we:

- Developed a consistent format. All man pages have the same information in the same order and under the same heading, so, for example, you can always find all the options under the "Options" heading, rather than scattered through the page.
- Added examples to the man pages. These examples show how the commands should be used without options, and with some of the more commonly-used options.
- Improved the process for getting online information from the man pages. Users can get an entire man page, or they can use the **help** command to reference various parts of the man page.
- Created a way to cross-reference within the online help system from one command to a related command (through the "See also" field of the man pages).

2.2.2 Improved Online Man Pages Besides the structural improvements we made to the man pages (standardized format, adding examples), the engineering group made some improvements to the access methods.

The **help** command not only offers help on topics (as discussed earlier), but has an extensive menu-driven interface for finding exactly the information in the man page that you want.

This interface lets you:

- Specify the section of the man page you want to see: Synopsis, Description, Options, Examples, Files, Variables, Return Value, Caveats, or See Also.
- Perform stack functions on the pieces of man page you are looking at, including pushing onto and popping off of the stack and swapping the current visible part of the man page with the top item on the stack (this feature is explained in more detail later).
- Move to another man page from the See Also list while keeping your place on the current man page so that you can return to it.
- "Scribble" on the online man page by creating a note file that then appears each time you (and only you) look at the man page in the future, whether with **help** or **man**.

To use the command help feature, type in **help** and the command name. For example, **help more** gets you the help for **more**. You can also specify a section of the manual just like you can with the **man** command. For example, **help 8 shutdown** gets you the **shutdown** command from section 8, not the **shutdown** system call from section 2.

The command help screen is different from the topic help screens. Figure 7 shows a typical command help screen. The commands you would enter to get this screen are:

```
help netconfig      <--- to call up the netconfig man page.
op                  <--- to display the netconfig options.
pu                  <--- to push the visible page onto the stack.
```

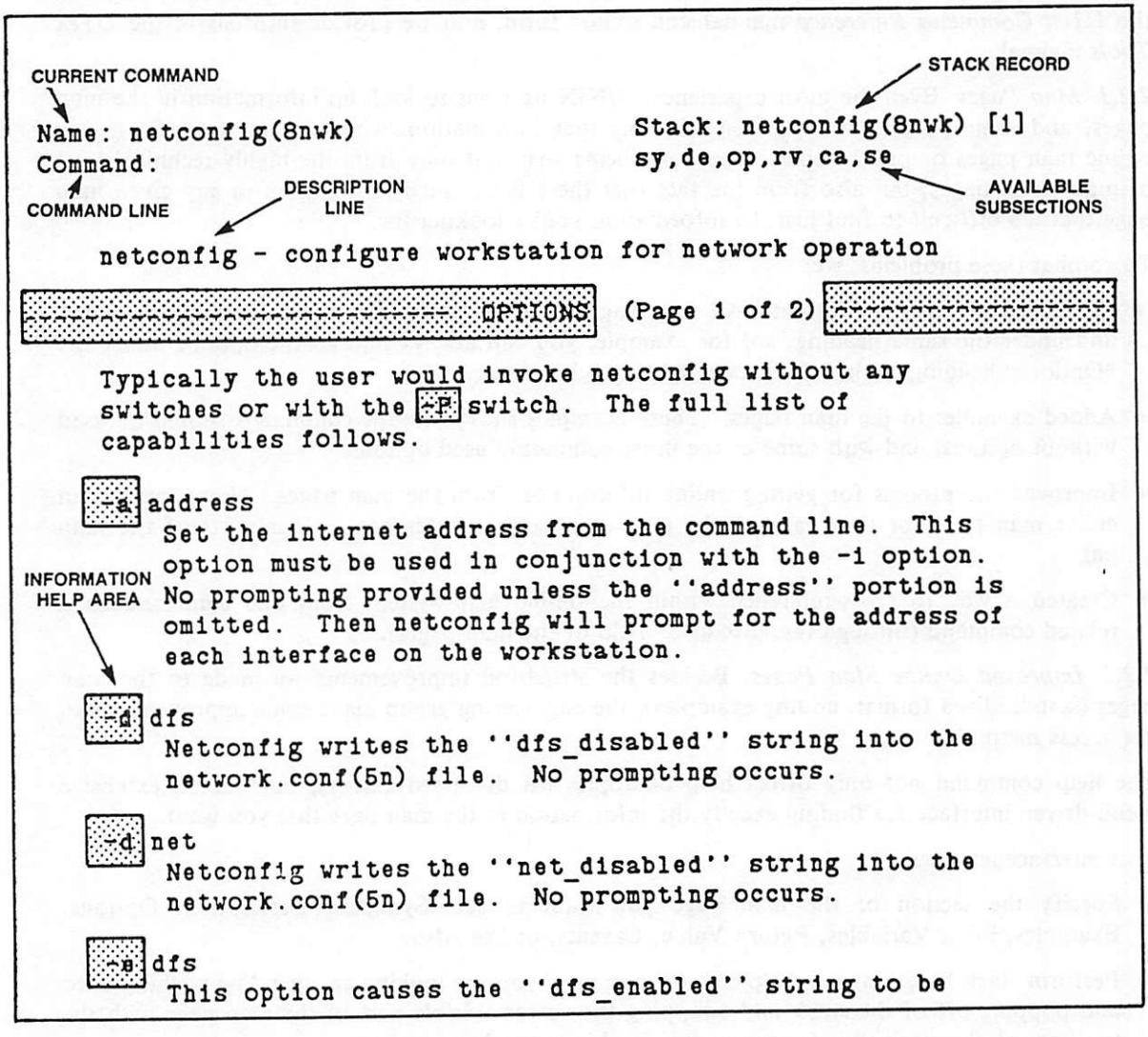


Figure 7. Typical Command Help Screen

During the first time you use the command help feature, you'd want to list all your choices. Type ?, and the following list of commands appears on the screen:

```
Section commands

sy  View SYNOPSIS section.
de  View DESCRIPTION section.
op  View OPTIONS section.
ex  View EXAMPLES section.
fi  View FILES section.
va  View VARIABLES section.
rv  View RETURN VALUE section.
di  View DIAGNOSTICS section.
ca  View CAVEATS section.
se  View SEE ALSO section.
re  View REFERENCES section.
no  View notes on current entry.

Movement commands

⟨CR⟩  Go to next section in order given above.
-     Go to previous section in order given above.
⟨sp⟩  Go to next page in current section or next section if last page.
^F    Go forward one page.
^B    Go back one page.
^D    Go forward one half-page.
^U    Go back one half-page.
1G    Go to first page in section.
G     Go to last page in section.
^L    Redraw screen (also applies in Help screen).

Notes commands

an  Update notes on current entry.
en  Edit notes on current entry.

Stack commands

pu  Push current page/section on stack.
po  Pop stack. Top element becomes current page/section.
sw  Swap current page/section for top of stack.
cs  Clear stack.
ps  Print stack values.

Other commands

?       Describe commands and error messages (this screen).
q⟨CR⟩   Quit.
he      Help. New page and section are prompted for.
```

Typing ? also lists and explains the error messages you can get while in the help interface.

2.2.2.1 See Also and Stack Functions You can move from the help display of one command to the help display of another command by using the *he* (for *help*) command of the help interface. The interface prompts you for a command name, and you can enter it in one of three ways:

- Just the command name. For example, **ls**.
- The section and the command name, as you might with the **man** command. For example, **1 more**.
- The command name and section in the form *command(section)*. For example, **shutdown(8)**.

Then, when you quit the second help, the first help reappears on the screen at the place you left it.

A stack is provided to give you a way to compare various pieces of information with one help command. You can use the stack commands to store more than one location in the help information for a command so you can come back to it later, and to switch back and forth between the page stored at the top of the stack and the current page on the help screen.

When there's information in the stack, the *stack record* appears in the upper right corner of the command help screen (see Figure 7). Figure 8 shows how to interpret the stack record.

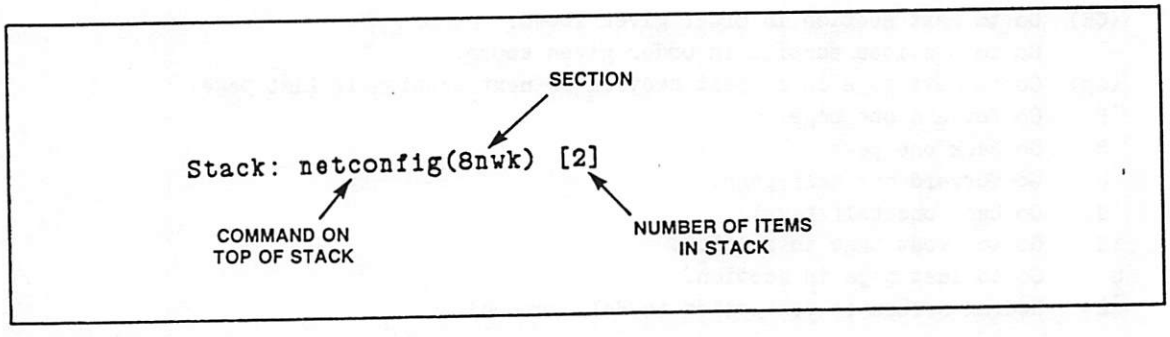


Figure 8. Stack Record

Figure 9 goes through a help session that gets information for two commands, **hostname** and **netconfig**, making use of the stack to swap between the help information for each command. The first column lists the commands you type, the second column lists what's in the help information area, the third column shows the contents of the stack, and the fourth column shows the stack record for the current stack.

YOU TYPE	HELP INFORMATION SCREEN CONTENTS	STACK CONTENTS	STACK RECORD
help netconfig	-----	-----	-----
de	*netconfig(8)* description	-----	-----
pu	*netconfig(8)* description	netconfig(8) description	`netconfig(8nwk) [1]`
op	*netconfig(8)* options	netconfig(8) description	`netconfig(8nwk) [1]`
pu	*netconfig(8)* options	netconfig(8) options netconfig(8) description	`netconfig(8nwk) [2]`
he	*netconfig(8)* options	netconfig(8) options netconfig(8) description	`netconfig(8nwk) [2]`
hostname	-----	netconfig(8) options netconfig(8) description	`netconfig(8nwk) [2]`
de	*hostname(1)* description	netconfig(8) options netconfig(8) description	`netconfig(8nwk) [2]`
sw	*netconfig(8)* options	hostname(1) description netconfig(8) description	`hostname(1n) [2]`
po	*hostame(1)* description	netconfig(8) description	`netconfig(8nwk) [1]`
po	*netconfig(8)* description	-----	-----

Figure 9. Using the Stack Functions

2.2.3 Scribbling in the Margins Often you want to make notes on a man page, write down things that happen when you use the command, remind yourself of features or "gotchas," and so on. Unless you have the source code for the man pages available, you usually cannot make these kinds of changes to the online documentation. But with the UTek **help** command, you can make a file for these notes with a simple command.

To do this, use the **en** command within the help interface. This puts you in the editor set by your EDIT environment variable, and you can write comments to your heart's content.

These comments go into a directory in your home directory called *.helpnotes*, created automatically the first time you use this feature. From then on, every time you use **help** or **man**, the file is read, and the information in it appears in the (new) Notes section at the end of the man page.

2.2.4 Write Your Own Man Pages Traditionally, UNIX sites obtained source code for the **man** command, and any local needs were hard-coded into the **man** source. Because we don't provide source code with our system, providing the flexibility of source code manipulation was achieved in the following ways:

- A superuser can map the binary directories that hold commands (*/usr/local*, */usr/public*, and so on) with directories in the */usr/man* directory. By editing the file */usr/lib/man/directories*, the superuser can make sure that any new man pages get into the search path. Figure 10 shows how */usr/lib/man/directories* maps binary directories to man page directories.

```
#
# Places to look for manuals.
#
# Man Directory          Binary directory        Operations
#
/usr/man                 /bin                    fiw
/usr/man                 /usr/bin
/usr/man                 /etc
/usr/man/man1            /usr/local              fiw
/usr/man/mann            /usr/new                fiw
/usr/man/manp            /usr/public             fiw
```

Figure 10. Sample /usr/lib/man/directories File

The Operations field of the file contains information on how the man pages in the directories should be formatted by **catman**

- Often there are local commands whose man pages aren't in the standard set. Traditionally, these pages have gone into */usr/man* subdirectories. Then, the superuser modified the **man** command source code to read the subdirectories. With UTek, you can not only use the */usr/lib/man/directories* file to map these new commands to new subdirectories of man pages(discussed above), but each of these new */usr/man* subdirectories is structured to look like the */usr/man* directory, with separate sections, so that the search algorithm is the same for all man pages.

- Users can set up a *.manrc* file in their home directory to designate a directory of personal man pages, man pages that they want ignored when they use the **man** command, and the order in which they want the man page sections searched when they use the **man** command. For example, users can designate that they don't want to see FORTRAN subroutine call man pages, so that this section doesn't even get searched during a **man** command, and that they want section 8 searched before section 2, and so on. Figure 11 shows a sample *.manrc* file.

```
personal : $HOME/man
ignore   : 3f 4+
sections : 1 1sh 1+ 8+ 3+ 2+ 5+ 7+ 6+
```

Figure 11. Sample .manrc File

The list of sections that users can choose to order or delete from their man search path is in */usr/lib/man/sections*. A superuser can add sections to or delete sections from this file to reflect the true state of the man pages on a given system.

2.2.5 UTek Tools Besides improving the man pages and access to man pages, we improved the organization and writing for the standard tutorial documents and wrote some new documents especially for our system.

For example, the following documents were either totally rewritten or written from scratch to cover features of UTek that were either not part of Berkeley UNIX (and therefore not part of Berkeley documentation), or that we felt needed different treatment than was given to them by Berkeley UNIX documenters:

- MH Mail
- MDQS
- C-Shell
- Distributed File System
- Ex and Vi
- ADB
- SDB
- RCS
- Make
- Using Make and RCS Together

3. SYSTEM ADMINISTRATION

Most people are not born with the knowledge it takes to administer a UNIX system. In fact, most UNIX users never learn how to administer the system. Traditionally, one or two people are given the root password and the charter to keep the system healthy and tame. And if *they* got in trouble, they had a guru to turn to.

However, the 6000 Family workstations were not designed to go into an environment that had or could afford to hire a guru. This meant that unless we wanted to spend all our time on the phone to customers, we had to develop a way to make system administration understandable and palatable.

3.1 Installation Information

We were told that the following scenario was likely, and that we had to develop our installation documentation in such a way as to make the installation of the workstation seem as simple and non-intimidating as possible. To keep the cost of the workstation down, and because the workstation is so simple to install, no field service representative automatically comes to install the system. (Installation service is available, as is service to deal with any unlikely problems with the installation.)

A customer orders a 6000 Family workstation, and it arrives at the customer's location in a box labeled *Tektronix*. The new owner unpacks the box, and inside needs to find everything necessary to install the workstation, including the knowledge.

Besides the workstation, the box contains the parts list, the standard manual set, and the Doc Map, discussed earlier. This map directs the owner to the *Installation Guide* for the workstation.

This *Installation Guide* was written especially for the person who has to install the workstation. It takes the installer through all the steps from unpacking the workstation to pressing the Start button for the first time. Then, it points to the *System Administration* manual for a detailed description of the first time start-up session.

The *System Administration* manual takes the owner through the the first time start-up session. Then it begins to teach system administration, simple but necessary things like setting the root password and the time and date.

3.2 Administering the System

Once the workstation is set up and running, someone has to make sure it stays that way. There are a number of tasks that have to be performed even if the workstation will have only one user, but that especially need doing if more than one person is going to use the system.

Because we couldn't count on the level of knowledge in any of our users, we had to cover all the bases, from instructing a novice about both UNIX and system administration to giving the gurus the few needed hints so that they can learn the workstation quickly and then be on their way.

Teaching a user UNIX was the responsibility of the documentation discussed in the first section of this paper. The *System Administration* manual's job is to teach the user: (1) what it means to be a system administrator, and (2) how to administer the workstation. We also knew we'd have some users who already knew about administering a system (maybe even a UNIX system), so we had to give them what they needed so that they could skip over what they already knew.

The *System Administration* manual has three distinct parts:

Background	How things work, like the *passwd* and *group* files, reasons for doing certain things certain ways, a few more details for some of the more esoteric parts of the UTek system, like Local Area Networks. Background information of this sort is also available for some aspects of the system in the *UTek Tools* manuals. The background information in the *System Administration* manual was put there because every system administrator would need it, while the information in the *UTek Tools* book is more specialized.
How-to	Detailed methods for performing system administration tasks. This part of the manual closely follows the *sysadmin interface* for the tasks covered by that interface, and has procedures for performing tasks that are not covered by the interface. (More on the sysadmin interface later.)
Troubleshooting	What to do if things start to go wrong. This part covers everything from emptying out file systems that get too full to reformatting the Winchester disk and rebuilding the system.

3.2.1 Sysadmin Interface Much of the how-to information in the *System Administration* manual describes how to use the *sysadmin interface*, an online, menu-oriented interface that makes tasks like creating a user account as easy as filling out a form online. (The system then automatically updates the proper system files.) It was provided to make a number of system administration tasks simpler:

- Adding, maintaining, and deleting user accounts.
- Adding, maintaining, and deleting groups.
- Performing system backups and restores.
- Adding applications and optional software.
- Configuring queues for devices.
- Setting network parameters.
- Configuring RS-232-C ports.
- Writing a message of the day.
- Setting up the table of system daemons.
- Configuring the intermachine mail system.

The interface lets you fill out forms for these tasks with the information you want, like creating print queues, assigning names and baud rates to new ports, or creating groups and assigning users to them. You never have to directly edit the files that are usually involved in these sorts of tasks, a task that involves intimate knowledge of the files' structure and can be confusing to anyone, especially a new user.

For example, adding a user to a UNIX system usually includes editing the */etc/passwd* file, creating a user's directory, putting .cshrc, .login, and/or .profile files into the user's directory, and **chown**ing the directory and files to the user. With the sysadmin interface, you use the User Login Account Maintenance menus to enter user-specific information (like login name and userid), and the sysadmin interface handles the rest; it edits the passwd file, creates the user's home directory, creates the login files in the home directory, and **chown**s everything. All the user needs to do is log in and set a (new) password. Figure 12 shows the first menu of the User Login Account Maintenance menu set.

```
                    User Account Maintenance

              1: (L)ist Usernames
              2: (F)ull User Information Listing
              3: (D)elete a User
              4: (C)hange User Information or Add User

            Choose one of the above ->
            (S)ave and go back, (Q)uit, <ESC> Back, (?,H)elp
   [ No changes made ]
--------------------------------------------------------------------------
```

Figure 12. User Accounting Interface

4. CONCLUSION

In this paper, we've shown how Tektronix tried to solve some UNIX documentation problems: avoiding initial intimidation, helping the UNIX novice *and* the UNIX expert through tutorials and various levels of online help, and making system administration tasks easier for the average person to understand.

Although we may not have solved all the UNIX documentation problems in the most efficient or best way, we feel we've provided better beginner and system administrator information than is usually available, and that we've improved access to the basic UNIX documentation set. We hope you can use our efforts as a stepping stone to improving your own UNIX documentation.

MH.5:
How to process 200 messages a day and still get some real work done[⋈]

Marshall T. Rose
Member, Research Technical Staff
Northrop Research and Technology Center[†]

John L. Romine
Computing Support Group
Department of Information and Computer Science
University of California, Irvine[‡]

ABSTRACT

The UCI version of the Rand Message Handling System (MH) is discussed. MH is a powerful user agent which operates in the ARPA Internet and UUCP environments. In addition to the usual functions provided by similar programs, MH has several distinguishing characteristics which give the user additional message handling capability. In particular, MH provides mechanisms for maintaining an organized mail environment, tailoring its behavior, and extending its functions.

This document describes MH from several perspectives. Particular emphasis is given to: the MH user environment, advanced features of MH which have proven to be particularly useful for sophisticated users of electronic mail, the user interface issues in MH, and the mh.5 distribution. The paper concludes with a summary of the authors' experiences with MH, and a discussion of future areas of enhancement.

[⋈] Alternate title: *MH: Your Key to Success.*

[†] One Research Park, Palos Verdes Peninsula, CA 90274. Telephone: 213/377-4811.
Computer mail: MRose%NRTC@USC-ECL, ...!{ucbvax!ucivax,trwrb}!nrtc!mrose.
[‡] University of California at Irvine, Irvine, CA 92717. Telephone: 714/856-6852.
Computer mail: J-Romine@USC-ECL, ...!{ucbvax,trwrb}!ucivax!jromine.

Contents

This document (version #1.43) was TEXset April 30, 1985 with DISS.STY v101.

MH.5:
How to process 200 messages a day and still get some real work done

Introduction

The UCI version of the Rand Message Handling System, MH, is a software system that performs two functions: first, it interfaces a user to a message transport system, so the user may receive and send mail; second, it permits the user to maintain an organized mail environment to facilitate the composition of new messages and the reading of old messages. In short, while not responsible for the delivery of messages, MH aids the user in handling mail.

MH was originally developed by the Rand Corporation, and initially was proprietary software. The Department of Information and Computer Science at University of California, Irvine, shortly after joining the Computer Science Network (CSnet), acquired a copy of MH, and began additional development of the software. Since that time, the Rand Corporation has declared MH to be in the public domain, and the UCI version of MH has passed through four major releases. The current version, mh.5, is available from U.C. Irvine for a nominal distribution fee, or may be retrieved from the University of Delaware via anonymous FTP.

Much credit must be given to the initial designers and implementors of MH: Bruce Borden, Stockton Gaines, and Norman Shapiro. Although MH has suffered significant development at UCI since Rand's initial release, the fundamental concepts of MH's environs have remained nearly unchanged. In addition, the authors of the current release gratefully acknowledge the comments of the many sites which have run various releases of MH in the past. In particular, the dozen or so beta test sites for mh.5 provided tremendous help in stabilizing the current release.

MH runs on different versions of the UNIX[1] operating system (such as Berkeley 4.2BSD and various flavors of v7). In addition, MH supports four different message transport interfaces: SendMail[EALLM83], the standard mailer for 4.2BSD systems; MMDF[DCROC79] and MMDF-II[DKING84], the Multi-Channel Memo Distribution Facility developed by the University of Delaware which forms the software-backbone for CSnet[DCOME83] mail relay service; SMTP, the ARPA Internet Simple Mail Transfer Protocol[SMTP]; and, a stand-alone delivery system.

[1] UNIX is a trademark of AT&T Bell Laboratories.

This paper is organized in a straight-forward fashion: Initially, the MH philosophy of mail handling is presented, along with a description of the environment which the MH user is given to process mail. Following this, certain advanced features of MH are discussed in more detail, such as facilities for selecting messages, and "advanced" concepts in *draft* handling. In addition, user interface issues in mail handling are addressed, and the merits of MH's approach is critically examined. Next, the mh.5 distribution package is described. Finally, we conclude by discussing the authors' experience with MH development and introducing areas where MH may be further developed.

Although familiarity with MH is not assumed on the part of the reader, some knowledge of the UNIX operating system is useful. Appendix A gives a short synopsis of the MH commands.

The MH Philosophy

Although MH has many traits which tend to distinguish it from other systems which handle mail, there is a single fundamental design decision which influences the interface between MH and the user: MH differs from most other systems in that it is composed of many small programs instead of one very large one. This architecture gives MH much of its strength, since intermediate and advanced users are able to take advantage of this flexibility.

The key to this flexibility is that the UNIX shell (usually the *C* shell or the *Bourne* shell), is the user's interface to MH. This means that when handling mail, the entire power of the shell is at the user's disposal, in addition to the facilities which MH provides. Hence, the user may intersperse mail handling commands with other commands in an arbitrary fashion, making use of command handling capabilities which the user's shell provides.

Furthermore, rather than storing messages in a complicated data structure within a monolithic file, each message in MH is a UNIX file, and each folder (an object which holds groups of messages) in MH is a UNIX directory. That is, the directory- and file-structure of UNIX is used directly. As a result, any UNIX file-handling command can be applied to any message.

To the novice, this may not make much sense or may not seem important. However, as users of MH become more experienced, they find this capability attractive. In addition, this approach is often quite pleasing to system implementors, because it minimizes the amount of coding to be performed, and given a modular design, changes to the software system can be maintained easily. There are, however, performance penalties to be paid with this scheme. This issue is considered later in the paper.

Having described how MH fits into the UNIX environment, we now discuss the mail handling environment which is available to the MH user.

The MH *Environs*

In the $*HOME* directory of each MH user, a file named *.mh_profile* contains static information about the user's MH environment, and default arguments for MH programs. For the latter case, each line of profile takes the form:

```
program-name:␣options
```

Each MH program consults the user's *.mh_profile* for its options. These options are consulted prior to evaluating any command-line arguments, and so provide the MH user the capability to customize the defaults for each command. Futher, by using the UNIX link facility, different names can be given to the same command. Since each MH command looks in the *.mh_profile* for a component with the name by which it was invoked, it's possible to have different defaults for the same program. For example, it is not uncommon to link *prompter* (a simple prompting editor front-end) under the name *rapid* in the user's *bin/* directory, and add to the *.mh_profile*:

```
rapid:␣-prepend␣-rapid
```

As a result, when *prompter* is invoked as *rapid*, it automatically uses the `'-prepend'` and `'-rapid'` options.

The profile component `''Path:''` is the path to the user's MH-directory, usually *Mail/*. In addition to containing the user's folders, the MH-directory also contains *skeletons* and *templates* used by the MH programs, and the user's *context* file. This latter file has the same format as the user's *.mh_profile*, and contains the dynamic, context-dependent information about the user's environment. Whenever MH looks for an MH-specific file, such as a template or skeleton, it first consults the user's MH-directory, and then a system-wide library area.

The MH user always has a *current folder*, which is the folder in which the user is currently (or was last) working. Since any MH program which deals with folders implicitly manipulates this information, the name of the current folder is stored in the *context* component `''Current-Folder:''`. Every folder has a *current message* known as `'cur'`. These values are the defaults for MH commands which accept folder and/or messages arguments.

MH programs make use of a set of envariables which further customize their behavior. The $*MH* envariable, if present, specifies the name of an alternate profile for the user. This allows a user of MH to easily maintain multiple mail-handling environments.

In terms of command syntax, most MH commands accept an optional *folder* argument, such as `'+outbox'`. Unlike most UNIX commands, all MH commands have switches which are words, rather than single letters. Switches may be abbreviated to the least unambiguous prefix. All MH commands also support

```
% inc
Incorporating new mail into inbox...

   1+ 03/16 Rand MH System    MH transcript  <<Here's the body of a sample m

% show
(Message inbox:1)
To: jromine@uci-icsa
Subject: MH transcript
Date: 16 Mar 85 18:28:59 PST (Sat)
From: Rand MH System <mh@uci-icsa>

Here's the body of a sample message.
% repl
To: Rand MH System <mh@uci-icsa>
cc: jromine@uci-icsa
Subject: Re: MH transcript
In-reply-to: Your message of 16 Mar 85 18:28:59 PST (Sat).
--------
Thanks for the test.

/JLR
^D

What now? send
% comp
To: MRose@UCI
cc:
Subject: sample comp
--------
Here's a sample compose for the MH transcript.

/JLR
^D

What now? send -verbose
 -- Posting for All Recipients --
  -- Local Recipients --
  MRose: address ok
 -- Recipient Copies Posted --
Message Processed
```

Figure 1
An MH Session

a '-help' switch, which lists the syntax of the command along with available switches, and the version number of the command. Most MH commands also take a 'msg' or 'msgs' argument which takes the form of a message number (''1''), a message range (''1-2''), a standard sequence name (''cur''), or a user-defined sequence name (''select'').

An MH *Transcript*

Figure 1 contains a transcript of a simple MH session. First, *inc* is run to incorporate the new mail into the user's ``+inbox'' folder.

A *scan* listing of the mail is printed while it is being incorporated. (The user could run *scan* explicitly to generate additional *scan* listings later on.) The *scan* listing gives the message number, followed by the date, message sender, and subject. (If the message originated from the user generating the listing, the ``to:'' addressee is displayed instead of the sender.) If the subject is short, the first part of the message body is displayed after the characters ``<<''. The plus sign ('+') after the message number indicates the current message.

The user *shows* the message, and decides to *reply*. A reply draft is created using the headers of the message being replied-to, using the default *replcomps* template. The default editor, *prompter*, is called to edit the draft. When an EOT is typed, *prompter* exits and the user is left at the What now? prompt. The option *send* is chosen. Since there were no problems in posting the draft with the message transport system, no additional output is produced. (MH is not verbose by default.)

The user then decides to compose a new message. The default skeleton, *components*, is copied to the draft, and *prompter* is once again called. After entering the addresses, subject, and body, the user then *sends* the *draft* from the What now? prompt, using ``send␣-verbose'', which causes MH to list out the message addresses as it submits them to the message transport system.

Some MH Features

We now consider certain advanced features in MH. These features have been chosen to demonstrate some useful capabilities available to the MH user.

Message Sequences and Selection

MH has several built-in message sequence names, which may be used anywhere a 'msg' or 'msgs' argument is expected. These are: 'cur', 'next', 'prev', 'first', 'last', and 'all'. Message ranges may also be specified. For example, 'all' is actually 'first-last', and '+mh␣last:5' references the last five messages in your '+mh' folder. A powerful capability of MH is the ability to use not only the pre-defined message sequence names, but also arbitrary user-defined message sequence names.

Although all MH programs recognize user-defined sequences when appropriate, the *pick* and *mark* commands can create and modify user-defined message sequences. The *mark* command allows low-level manipulation of sequences, and is not particularly interesting in our discussion.

The *pick* command selects certain messages out of a folder. The criteria used for selection may be a search string and/or a date range.

Searching is performed on either a specific header in the message (e.g., ``To:''), or anywhere within the message. By default, *pick* lists out the message numbers that matched the selection criteria. Thus, *pick* is useful in backquoted operations to the shell. For example, to scan all the messages in the current folder from "frated", the MH user issues the command:

```
scan␣'pick␣-from␣frated'
```

To perform more complicated message selection, user-defined sequences are employed. Supplying a `'-sequence␣name'` argument to *pick*, will cause it to define the sequence `'name'` as those messages matched.

Giving *pick* a list of messages causes it to limit its search to just those messages. For example, to find all the messages in the current folder from "frated" also dated before friday:

```
pick␣-from␣frated␣-sequence␣select
pick␣select␣-before␣friday␣-sequence␣select
```

With the first *pick* command, the sequence ``select'' is defined to be all those messages from "frated". In the second command, only those messages already in the ``select'' sequence are searched, and the ``select'' sequence is redefined to be only those messages which are also dated before friday. Those messages could then be *shown* with:

```
show␣select
```

When a `'-sequence␣name'` argument is given to *pick*, the default behavior — listing the message numbers matched — is inhibited. To re-enable this behavior, the `'-list'` option may be given. As a result, advanced users of MH often put the following line in their *.mh_profile*:

```
pick:␣-sequence␣select␣-list
```

which allows them to easily make use of the `'select'` sequence as the messages last selected with *pick*.

Often it is desirable to act upon those messages which are *not* members of a given sequence. For this purpose, the ``Sequence-Negation:'' profile entry is useful. If the name of a user-defined sequence is prefixed with the value of the sequence-negation profile entry, MH commands will operate upon those messages which are *not* members of that sequence. For example, given a profile entry of:

```
Sequence-Negation:␣not
```

those messages which are not in the `'select'` sequence could be *scan*'d with:

```
scan␣notselect
```

Obviously, some confusion could result if an attempt was made to define a sequence name which began with the sequence-negation string (e.g., ``notselect''). For this reason, MH users will often use a single character, which their shell doesn't interpret, as their sequence-negation string (e.g., up-caret ('^') for *C* Shell users, and exclamation-mark ('!') for *Bourne* shell users).

MH also provides a way of automatically remembering the last message list given to an MH command. This facility is implemented by using a profile entry called ``Previous-Sequence:''.

Draft Handling

After the initial edit of a message draft, the *comp*, *dist*, *forw*, and *repl* programs give the user a `What now?` prompt. The valid responses include: *edit* to re-edit the draft, *quit* to exit without sending the draft, *send* to send the draft, and *push* to send the draft in the background.

When the *send* option is given, the draft is posted with the message transport system. If there problems posting the draft, the `What now?` prompt is re-issued, so errors in the draft may be corrected.

Since posting the draft can be slow, the *push* option allows the MH user to send the draft in the background, and return immediately to the shell. If there are problems posting the message, the user will not see the diagnostics produced by the message transport system. For this reason, if *push* is used instead of *send*, and the message is not successfully posted, MH mails a message to the user containing any diagnostics which the message transport system produced along with a copy of the message. Later, the draft may be re-edited by entering ``comp␣-use''.

A relatively new feature of MH is the ability to use a folder to store multiple drafts. These drafts are kept in an ordinary MH folder, and may be operated upon by MH commands. To enable this feature, the MH user selects a folder-name for the draft-folder, and creates an entry in the *.mh_profile*:

```
Draft-Folder:␣+foldername
```

From this point on, when a message is composed, the draft will be created as a message in that folder, instead of using the *draft* file in the user's MH directory. Unfortunately, if posting problems occur on a message which has been *push*'d, it may be difficult to re-edit the draft with ``comp␣-use''. This might be the case if the user had started composing another message, while that first draft was being posted. In that event, the current-message in the draft-folder would no longer point to the failed draft.

There is a solution for this problem, however. By default, *push* assumes the `'-forward'` option, which says that if the message draft fails to be posted, it

should be forwarded back to the user in the error report which *push* generates. The failed draft may then be extracted with the *burst* program (discussed later).

BBoards

MH has a convenient interface to the UCI BBoards facility[MRose84a].[2] This facility permits the efficient distribution of interest group messages on a single host, to a group of hosts under a single administration, and to the ARPA Internet community.

Although most readers are probably familiar with the concept of an interest group in the Internet context, a brief description is now given. Observant readers will notice that the distributed nature of the "network news" (a.k.a. USENET) tends to avoid many of the problems described below.

Described simply, an interest group is composed of a number of subscribers with a common interest. These subscribers post mail to a single address, known as the *distribution* address (e.g., `MH-Workers@UCI`. From this distribution address, a copy of the message is sent to each subscriber. Each group has a *moderator*, who is the person that runs the group. This moderator can usually be reached at a special address, known as the *request* address (e.g., `MH-Workers-Request@UCI`). Usually, the responsibilities of the moderator are quite simple, since the mail system handles distribution to subscribers automatically. In some interest groups, instead of each separate message being distributed directly to subscribers, a batch of (hopefully related) messages are put into a *digest* format by the moderator and then sent to the subscribers. (This is similar to a newsletter format.) Although this requires more work on the part of the moderator and introduces delays, such groups tend to be better organized.

Unfortunately, some problems arise with the scheme outlined above. First, if two users on the same host subscribe to the same interest group, two copies of the message are delivered. This is wasteful of both processor and disk resources at that host.

Second, some groups carry a lot of traffic. Although subscription to a group does indicate interest on the part of a subscriber, it is usually not interesting to get 50 or so messages delivered each day to the user's private maildrop, interspersed with *personal* mail, which is likely to be of a much more important and timely nature.

Third, if a subscriber's address in a distribution list becomes "bad" somehow and causes failed mail to be returned, the originator of the message is normally notified. It is not uncommon for a large list to have several bogus addresses. This results in the originator being flooded with "error messages" from mailers across

[2] The UCI BBoards facility can run under either the MMDF or SendMail, or in a more restricted form under stand-alone MH.

the Internet stating that a given address on the list was bad. Needless to say, the originator usually does not care if the bogus addresses got a copy of the message or not. The originator is merely interested in posting a message to the group at large. On the other hand, the moderator of the group does care if there are bogus addresses on the list, but ironically does not receive notification.

To solve these problems, the UCI BBoards facility introduces a new entity into the picture: a *distribution channel.* All interest group mail is handled by the special mail system component. The distribution address for an interest-group maps mail for that interest-group to the distribution channel, which then performs several actions. First, if local delivery is to be performed, a copy of the message is placed in a global maildrop for the interest group with a timestamp and a unique number. Local users can read messages posted for the interest group by reading this "public" maildrop. Second, if further distribution is to take place, a copy of the message is sent to the distribution address in such a way that if any of the addresses are bogus, failure notices will be returned to the local maintainer of the group address list, rather than the originator of the message.

This scheme has several advantages: First, messages delivered to the local host are processed and saved once in a globally accessible area. The UCI BBoards facility supports software which allows a user to query an interest group for new messages and to read and process those messages in the MH-style. Second, once a host administrator subscribes to an interest group, each user may join or quit the list's readership without contacting anyone. Third, a hierarchical distribution scheme can be constructed to reduce the amount of delivery effort. Finally, errors are prevented from propagating. When an address on the distribution list goes bad, the list moderator who is responsible for the address is notified. If a local moderator does not exist, then the local PostMaster is notified (not the global group moderator).

In addition to solving the problems outlined above, the UCI BBoards facility supports several other capabilities. BBoards may be automatically archived in order to conserve disk space and reduce processing time when reading current items. Also, the archives can be separately maintained on tape for access by interested researchers.

Special alias files may be generated which allow the MH user to shorten address entry. For example, instead of sending to `SF-Lovers@Rutgers`, a user of MH usually sends to `''SF-Lovers''` and the MH aliasing facility automatically makes the appropriate expansion in the headers of the outgoing message. Hence, the user need only know the name of an interest group and not its global network address.

Finally, the UCI BBoards facility supports *private* interest groups using the UNIX group access mechanism. This allows a group of people on the same or different machines to conduct a private discussion.

The practical upshot of all this is that the UCI BBoards facility automates the vast majority of BBoards handling from the point of view of both the PostMaster and the user.

MH provides three programs to deal with interest groups. The *bbc* program is used to check on the status of one or more groups, and to optionally start an MH shell on those groups which the user is interested in. The *bbl* program can be used to manually perform maintenance on a discussion group beyond the normal automatic capabilities of the UCI BBoards facility. Finally, the *msh* program implements an MH shell for reading BBoards, in which nearly all of the MH commands are implemented in a single program.

Observant readers may note that the use of *msh* is contrary to the MH philosophy of using relatively small, single-purpose programs. Sadly, the authors admit that this is true. In an effort to minimize use of system resources however, BBoards are kept in maildrop format instead of folders.[3] Some research has gone into overcoming this problem to restore MH's purity of purpose, but all solutions proposed to date are either unworkable or require significant recoding of MH's internals.

Bursting

Internet interest group mail is often sent out in digest form. The experienced MH user may wish to deal with the digest messages on an individual basis, however. The *burst* program allows the MH user to extract these digest messages, and store each as an individual MH message.

Burst will also extract forwarded messages generated by *forw* (or the forwarded message in the error report generated by *push*, as described above). Although *burst* cannot always decapsulate messages encapsulated by sites not running MH, it adheres to the proposed standard described in [MRose85b].

[3] When the message transport system delivers a message to a user it stores it in a single file, called a *maildrop*. Since many messages may be present in a single maildrop, (in theory) there is a unique string acting as a separator between messages in the maildrop. Although this is convenient for storage of messages, it makes retrieval more difficult unless a separate index into the maildrop is kept. This latter approach is taken by the *msg* program available with MMDF-II and by *msh* as well.

Distributed Mail

The ARPA Internet community consists of many types of heterogeneous nodes. Some hosts are large mainframe computers, others are personal workstations. All communicate using the MILSTD TCP/IP protocol suite[IP, TCP]. Messages which conform to the Standard for the Format of ARPA Internet Text Messages[DCROC82] are exchanged using the Simple Mail Transfer Protocol[SMTP].

On smaller nodes in the ARPA Internet, it is often impractical to maintain a message transport system (e.g., SendMail). For example, a workstation may not have sufficient resources (cycles, disk space) in order to permit an SMTP server and associated local mail delivery system to be kept resident and continuously running. Furthermore, the workstation could be off-net for extended periods of time. Similarly, it may be expensive (or impossible) to keep a personal computer interconnected to an IP-style network for long periods of time. In other words, the node is lacking the resource known as "connectivity".

Despite this, it is often desirable to be able to manage mail with MH on these smaller nodes, and they often support a user agent to aid the tasks of mail handling. To solve this problem, a network node which can support a message transport entity (known as *service* host) offers a maildrop service to these less endowed nodes (known as *client* hosts). The Post Office Protocol[JREYN84] (POP) is intended to permit a workstation to dynamically access a maildrop on a service host to pick-up mail.[4] The level of access includes the ability to determine the number of messages in the maildrop and the size of each message, as well as to retrieve and delete individual messages. More sophisticated implementations of the POP server are able to distinguish between the header and body portion of each message, and send n lines of a message to the POP client. This capability is useful in thinly connected environments where conservation of bandwidth is important. By utilizing a more intelligent POP client, a user may generate "scan listings" and decide dynamically which messages are worth taking delivery on. The philosophy of the POP is to put intelligence in the POP clients and not the POP servers.

The current release of MH supports the above model fully. A POP client program is available to retrieve a maildrop from a POP service host. In addition, using the SMTP configuration for delivery in MH (either in conjunction with SendMail or the MMDF), a user is able to specify a search-list of service hosts (and/or networks) to try to post mail. Using this search-list, when an MH user posts a draft, the *post* program will attempt to establish an SMTP connection with each host in the search-list to post the message until it succeeds. Initial

[4] Actually, there are three different descriptions of the POP. The first, cited in [JREYN84], was the original description of the protocol, which suffered from certain problems. Since then, two alternate descriptions have been developed. The official revision of the POP[MBUTL85], and the revision of the POP which MH uses (which is documented in an internal memorandum in the MH release). This paper considers the POP in the context of the MH release.

experimentation using the POP and MH in a local network environment has proved quite successful.

User Interface Issues in MH

At this point, it is perhaps useful to take a step backwards and examine the success and problems of MH's approach to user interfaces.

Creeping Featurism

A complaint often heard about systems which undergo substantial development by many people over a number of years, is that more and more options are introduced which add little to the functionality but greatly increase the amount of information a user needs to know in order to get useful work done. This is usually referred to as *creeping featurism.*

Unfortunately MH, having undergone six years of off-and-on development by ten or so well-meaning programmers (the present authors included), suffers mightily from this. For example, the *send* command has twenty-five visible switches, and at least nine hidden switches, for a total of thirty-four. The poor user who types

```
send␣-help
```

watches the options scroll off the screen (since the `'-help'` switch also lists out four other lines of information).[5] The sad part is that all of these switches are useful in one form or another.

There are a lot of good things to be said for the "one program, one function" philosophy of system design. In the MH case, however, each program really does only one mail handling activity (with a few minor exceptions). The options associated with each command are present to modify the program's behavior to perform similar, but slightly different tasks. In further defense of MH, note that there are 32 MH commands at present, all performing different tasks.

The problem with creeping featurism though, is that while the functionality of the system increases sub-linearly, the complexity of the system increases linearly. That is, although the number of switches that a program takes might double, it is unlikely that the program's functionality or capabilities will double.

---

[5] Recently, this was fixed by compressing the way in which switches are presented. The solution is only temporary however, as *send* will no doubt acquire an *endless* number of switches in the years to come.

```
To:
cc:
Bcc:
Fcc: outbox
Fcc:
Subject:
Reply-To:
--------
```

Figure 2
Draft Skeleton

```
To: <reply-to|from>
cc: <?to|cc><to>,<cc>
Fcc: +outbox
Fcc: <?fcc><fcc>
Subject: <?subject>Re: <subject>
In-reply-to: <?date><?message-id>Your message of <date>.
     <message-id>
In-reply-to: <?date><!message-id>Your message of <date>.
--------
```

Figure 3
Reply Template

Templates versus Switches

One way to trim the explosion of available options, while still increasing functionality, is to introduce options with a richer domain. Hence, instead of using options which take *on* or *off* forms or simple numeric or string values, the possible values which an option might take on is given a large space. There are several ways that this might be accomplished.

The *comp*, *dist*, and *forw* programs use draft *skeletons* (simple form fill-in files) to construct the general format of the draft being composed. An example of a draft skeleton used for composing new messages (by *comp*) is shown in Figure 2. The approach is to let the user specify (and later edit) both arbitrary headers of draft and the body of the draft. Note while most of the fields are empty, the first ''Fcc:'' field already contains a value. By using the simple prompting editor, *prompter*, the user can speedily enter the headers of the message. The *prompter* program given the skeleton in Figure 2 would prompt the user for the contents of each field, except for the second ''fcc:'', which it would include verbatim. It would then read the body of the message up to an end-of-file. Naturally, the MH user is free to use *any* editor to edit *any* part of the draft (headers or body). This example demonstrates the flexibility achieved by not limiting what headers a draft may contain (which most mail sending programs do), while still retaining the simplicity of being able to treat the entire message draft as a UNIX file.

```
From: <?me>Message Agent \<<me>>
To: <reply-to|from>
Fcc: +rcvtrip
Fcc: <?fcc><fcc>
Subject: <?subject>BEEP! Re: <subject>
Subject: <!subject>BEEP!
In-reply-to: <?date><?message-id>Your message of <date>.
     <message-id>
In-reply-to: <?date><!message-id>Your message of <date>.
--------

    This is an automatic reply.  Feel free to send additional mail, as only
    this one notice will be generated.

    I am attending the USENIX Summer '85 conference in Portland, Oregon.
    I expect to be reading mail again on the 16th of June.

/mtr
```

Figure 4
The *tripcomps* Reply Template

Another more interesting approach is used by the *repl* command, which constructs a draft in reply-to a previously received message. Instead of adding switches to indicate which fields of the draft should be derived from the message being replied-to, and how they should be derived, a single option, the ability to specify a *template*, was made available. An example of a reply template is shown in Figure 3. Put simply, based on the presence of certain fields in the message being replied-to, and a few switches given by the user, using the reply template, *repl* generates the reply draft automatically.

This facility, for example, can be used to generate automatic replies.[6] One function might be to write a *rcvtrip* shell script which automatically answered messages when mail wasn't being read for a period of time (e.g., while attending a conference). An example of a reply template at the heart of such a script is shown in Figure 4.

Finally, another application might be to utilize the highly useful letter bomb protocol.[7] The important thing to note about this template is that it generates not only the headers of the reply draft (with a creative ``Reply-to:'' address), but the body as well. Hence, the commands

```
repl␣-form␣bombcomps␣-noedit␣;␣rmm
What␣now?␣push
```

[6] MH supports the notion of a user-defined *mail hook* which is invoked each time a user receives mail.

[7] The authors wish to credit Ron Natalie of the Ballistics Research Laboratory in Aberdeen, Maryland for formalizing the use of this protocol in the ARPA Internet community.

```
To: <reply-to|from>
cc: <?to|cc><to>,<cc>
Fcc: +outbox
Fcc: <?fcc><fcc>
Subject: <?subject>Re: <subject>
In-reply-to: <?date><?message-id>Your message of <date>.
     <message-id>
In-reply-to: <?date><!message-id>Your message of <date>.
Reply-To: /dev/null
--------
```

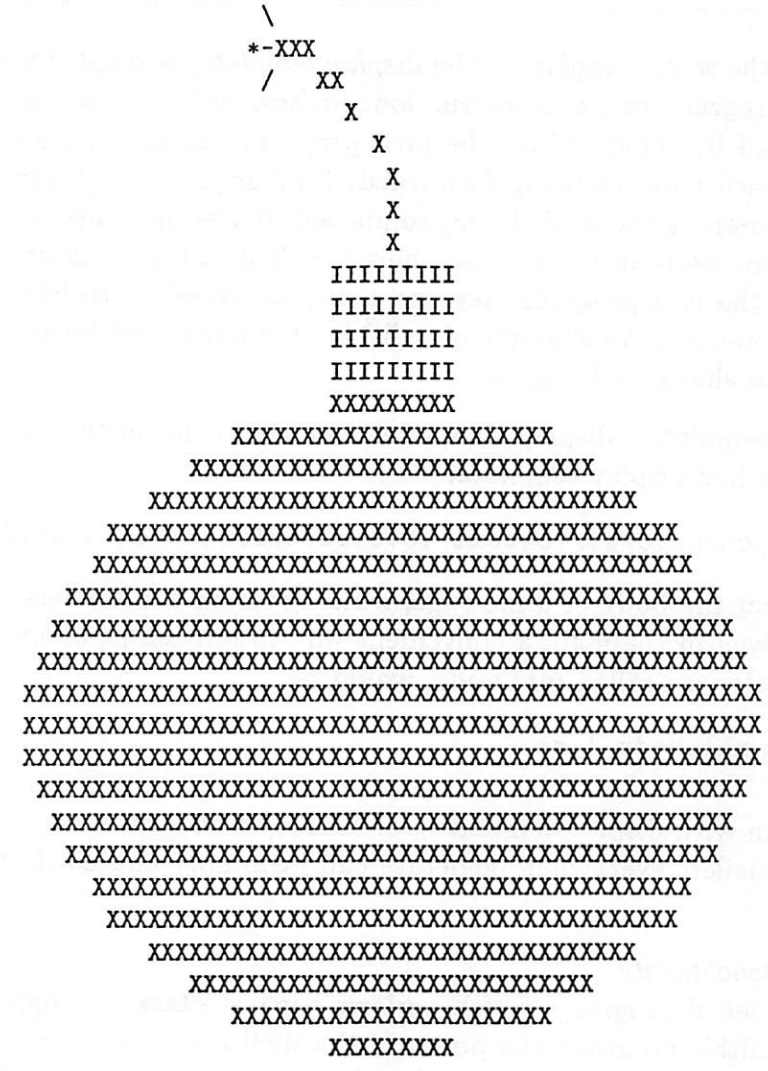


Figure 5
The *bombcomps* Reply Template

are very handy for dealing with disturbing mail in a straight-forward manner. Of course, *repl* could be linked to *bomb* in the user's *bin/* directory and an appropriate line could be added to the user's MH profile, in order to further shorten type-in.

```
width=80,length=0,overflowtext=,overflowoffset=10
Date:leftadjust,compress,compwidth=9
Subject:leftadjust,compress,compwidth=9
From:leftadjust,compress,compwidth=9
To:leftadjust,compress,compwidth=9
cc:leftadjust,compress,compwidth=9
Resent-Note:leftadjust,compwidth=9
:
body:nocomponent,overflowoffset=0
```

Figure 6
Display Template

A variation on the reply template is the *display template.* A display template, as used by the *mhl* program, contains instructions on how to format a message. In addition to being used by *show*, et. al., the *forw* program can also use a display template to format each message being forwarded. Similarly, although *repl* uses a reply template to construct the draft being composed, it also may use a display template to format the body of the message being replied-to for enclosure in the reply. Furthermore, the *post* program may use a display template to format the body of a blind-carbon-copy. An example of a display template used for formatting forwarded messages is shown in Figure 6.

As with reply templates, display templates can offer a lot of functionality. For example, the one line display template:

```
body:nocomponent,overflowtext=,overflowoffset=0,width=10000
```

can be used to extract the body of a message, while ignoring the headers. Hence, if a *shar* archive arrived in the mail, a convenient way to unpack it, assuming the above display template was called *mhl.body*, would be:

```
show␣-form␣mhl.body␣|␣sh
```

The biggest win with display templates, of course, is that all those annoying header lines which mailers everywhere generate can be simply and easily filtered out.

Modularity versus Monolithicity

Since MH is a set of programs which perform separate tasks, as opposed to being a single, monolithic program, the power of the shell is used directly to aid in mail-handling. One powerful capability which this design achieves is the ability to

```
: 'mpick - relate messages /mtr'
PATH=:/bin:/usr/bin:/usr/ucb:/usr/local:/usr/local/lib/mh; export PATH
F="" M="" S=""

for A in $*
do
    case $A in
        -*)     S="$S $A" ;;

        +*|@*)  case $F in
                    "")  F=$A ;;
                    *)   echo "mpick: only one folder at a time" 1>&2
                         exit 1 ;;
                esac ;;

        *)      M="$M $A" ;;
    esac
done

S="$S -sequence hits -list -nozero"

if mark $F all -add -sequence hits;
    then mark $F all -delete -sequence hits;
    else exit 1;
fi

for A in ${M-cur}
do
    for C in `mhpath $F $A`
    do
        if [ -r $C ];
            then
                I=`mhl -form mhl.msgid $C`;
                case $I in
                    "")  echo "no message-id in message `basename $C`" 1>&2 ;;
                    *)   pick --in-reply-to "$I" $S ;;
                esac
            else
                echo "message $A doesn't exist" 1>&2; exit 1;
        fi
    done
done

exit 0
```

Figure 7
The *mpick* Script

extend the MH command set, by developing shell scripts which use the standard MH programs to accomplish complicated or specialized tasks.

```
: 'append - stupid append editor for MH - /jlr'
case $# in
   1|2) case $# in
           1) F=$1; echo -n "Append file: " 1>&2; read A ;;
           2) F=$2; A=$1 ;;
        esac
        cat $A < /dev/null >> $F ;;
     *) echo "append: arg count" 1>&2 ; exit 1 ;;
esac
exit
```

Figure 8
The *append* Editor

For example, in the MH distribution there is a shell script called *mpick* (shown in Figure 7) which tries to locate all the messages which pertain to a given discussion, by looking at the ''Message-ID:'' and ''In-reply-to:'' headers, to find matching message-ids.[8]

Unfortunately, some parts of MH are somewhat monolithic. An example of this is the `What now?` prompt. There are only a few options at this prompt, and one cannot give a normal shell command. Some MH users seem to feel that more options should be added to the `What now?` prompt, such as an *insert-file* option. It was argued that just about any editor would allow you to insert a file, and another `What now?` option was not needed. These users persisted, however, so the problem was solved, by writing a trivial shell script "editor" (see Figure 8) which could be invoked by the *edit* option:

```
What now? edit append filename
```

A better interface at this point is really needed, however. One possibility is to simply pass any unrecognized commands on to a shell for interpretation, supplying the path name of the draft file as an argument. A solution which shows more promise is to give you a sub-shell *instead* of the `What now?` prompt, and setup certain envariables so that the MH commands would act upon the *draft* by default. For example, *show* with no `'msgs'` arguments would show the draft instead of the current message. This alternative has recently been implemented and is under testing.

[8] Note that the shell scripts included in the MH distribution are written for the *Bourne* shell, and have a ':' as the first character of the first line, so they will be portable to all versions of UNIX, not just those which support the Berkeley '#!' enhancement.

```
bin      /usr/local
bboards on
editor   /usr/local/prompter
etc      /usr/local/lib/mh
mail     /usr/spool/mail
manuals local
mts      sendmail/smtp
news     off
options BSD42
options MHE NETWORK
options UCI
```

Figure 9
Sample MH Configuration File

The MH Distribution

The mh.5 distribution is now briefly described, both in terms of static configuration methods and dynamic tailoring. Appendix B describes the mechanics of receiving an mh.5 distribution.

Configurable MH

The MH distribution currently runs on a large number of different UNIX versions, ranging from MicroSoft XENIX to Berkeley 4.2BSD. All the code which is specific to a particular target environment is enabled via the C-preprocessor ``#ifdef'' mechanism, so compilation under different versions of UNIX is trivial. There are, however, a large number of compile-time options which may vary from site to site, so an automated configuration method was needed.

The MH-installer must create a configuration file, which contains a list of the compile-time options and the values which are desired for them. Compile-time options include the installation location for MH, what kind of message transport system is to be used, and the default editor for the installation. An example of such a configuration file is shown in Figure 9.

After creating this file (several examples are included in the distribution), the installer runs the *mhconfig* program, which customizes the *Makefiles* and some of the programs, for that site's particular installation. No hand-editing of any source code should be necessary, under normal circumstances.

Interface to the Message Transport System

MH will run with a number of message transport systems, including SendMail, MMDF-II, and a small stand-alone system. One flexible method of posting mail is through an SMTP connection. There are a couple of major wins in using this configuration: First, none of the MH programs need to know where the interface programs to the message transport system are located, which makes them easier to move between systems. Second, mail can be posted on relay hosts, and the local

```
mmdfldir:       /usr/spool/mail
mmdflfil:
mmdelim1:       \001\001\001\001\n
mmdelim2:       \001\001\001\001\n
mmailid:        0
lockstyle:      0
lockldir:

hostable:       /usr/local/lib/mh/hosts
servers:        localhost \01localnet
```

Figure 10
Sample MTS Tailor File

host of an MH user may not need a message transport system at all (as alluded to in the preceeding discussion on the POP).

Those parts of MH which interact with the local message transport agent read additional tailoring information when they start.[9] This information includes the location of standard and alternate maildrops, maildrop delimiter strings, the locking directory and locking style, and other tailoring information specific for the particular message transport system in use (e.g., the default server search-list when mail is posted with the SMTP). In most cases, by using a tailor file, each site running a similar MH configuration is able to simply transfer MH binaries between hosts. An example of such a tailor file is shown in Figure 10.

A continuing question which is often raised is how intelligent should user agents (like MH and UCB *Mail*) be with respect to the environment in which they operate. At present, MH likes to determine the official hostnames for addresses when posting mail. Many argue that this is improper or unnecessary behavior for a user agent, and that the local message transport agent should handle these functions. Unfortunately, this implies that the message transport agent should munge headers when mail is posted to remove local host aliases and only permit address fields with fully-qualified addresses. Sadly, neither SendMail nor MMDF-II really gets this right (flames to */dev/null* please). The current MH maintainers believe that the resolution of host aliases to official names should be a well-supported interface with the local message transport agent. However, to provide equal time to those who hold opposite views, MH supports a configuration option called ''DUMB'' which disables MH's attempts to resolve addresses into fully-qualified strings.

[9] This simple facility is based on a more extensive tailoring capability found in MMDF-II.

Concluding Remarks

While MH has undergone significant development since the original Rand release, the authors have tried to keep the fundamental concepts of MH unchanged. The authors have continually had to battle against well-meaning MH users who wanted to make MH more like other (less powerful) user agents. More and more "features" were often suggested for MH, usually at the expense of making MH less general, and more specific. In nearly all cases, the "features" which these users wanted were already present in MH in a slightly different form, or could be realized by simply writing a short shell script. A classic example is the repeated requests by one user to have *dist* take a list of messages rather than a single message and distribute each one of them in turn. A simple shell script which called *dist* repeatedly, perhaps with "canned" arguments so the user typed in addressing information only once, would easily meet this request.

A number of MH comands have a large number of options. When adding options, the authors have tried to make the options general, while still accomodating the requests of specific users. An example of a specific request which was implemented as a general feature is the ``Previous-Sequence'' profile entry (mentioned above). If you use this profile entry, every MH command is forced to write out *context* changes, making every command somewhat slower. Since only a few users wanted this capability, it was implemented in such a way that users who didn't want it, didn't have to pay the cost of slowing down every MH command.

MH has a powerful tailoring capability provided by the *.mh_profile*. Using profile entries, users may customize their own environment without affecting others. Novice users often take advantage of the MH-tailoring capabilities to try to make MH work similarly to other user agents they've used. This has the advantage of allowing them to quickly begin using MH to handle their mail. However, since these novice users don't take advantange of all the capabilities of MH, they frequently will complain about things they think can't be done with MH, or could be done "better" some other way. Fortunately, as these users become more experienced with both MH and UNIX, they can modify their environment to take better advantage of all of MH's capabilities. Novice MH users who see features lacking are encouraged to take a better look at what MH *can* do, instead of trying to make MH into something it isn't. This may sound rather inflammatory, but it would really be a much nicer world for us all if users of software systems would read the manual prior to asking questions.

For a moment, let's consider the evolution of one MH feature which has proved itself to be very useful. As users began employing MH to handle their mail, the number of messages that could be processed in a given amount of time increased greatly. As the volume of messages increased however, it became clear that some MH operations were too slow, in particular the interaction with the (slow) message transport system. To overcome this problem, the *push* option was

added at the `What now?` prompt. Originally, this option was hidden from novice users and did little more than send the message in the background: any output generated by the background *send* process would be printed asyncronously on the terminal. If a message failed posting with the message transport system, it would simply be left in the *draft* file.

Gradually, other features were added to *push*. Since users wanted to be able to send more than one draft at a time, *push* was changed to optionally rename the draft file before posting it. (This is what the hidden `'-unique'` option does.) Having message transport system diagnostics written asyncronously on the user's terminal was annoying, so *push* was made to intercept these diagnostics, and mail the user a report containing them. Although the diagnostic report mailed back by *push* contains the name of the draft which failed, a useful added feature was the ability to have *push* include the failed draft as well. Eventually, the draft-folder mechanism was implemented to make handling multiple message drafts much easier.

TODO

There are, no doubt, a number of improvements which could be made to MH. At the present time, what further development should MH suffer? Although not by any means inclusive, here's a list:

1. Performance Enhancements
   Hardware gets faster all the time, but people always complain that software is too slow. Owing to its user interface style, MH is somewhat slower than monolithic programs like UCB *Mail*. It would be nice if MH could be tuned or accelerated somehow.

2. Port to System 5
   MH runs on 4.2BSD UNIX and Version 7 variants. It should not be difficult to port MH to a SYS5 environment. This should significantly increase the number of hosts on which MH can run. The authors, lacking a SYS5 machine (and experience with SYS5) to perform the port, are actively seeking a System 5 guru to attempt this feat.

3. Interface to the Network News
   Not all sites that run MH are in the ARPA Internet, and as such the UCI BBoards facility may not be of much use to them. A good MH interface to the network news would allow users on hosts with a news feed to employ the same interface for reading and sending both mail and news.

4. Programmed Instruction for Beginners
   The complexity of MH is often intimidating to new users. It would be

nice to develop a set of *learn* lessons for those users who don't like *man* pages and non-interactive tutorials.

5. Message List Expansion
   At present, when a list of messages is given to an MH command, it expands the list and processes each message in numerical order rather than the order in which the messages were given (e.g., ``show␣2␣1'' *show*s message 1 and then message 2). It would be nice if MH processed messages in the order they were given.

6. Context Changes
   In nearly all cases, an MH command does not write out context changes until it is about to exit successfully. There is some controversy as to whether this is the correct behavior in all cases. Some argue that once an MH command has fully parsed its argument list, the context should be updated.

REFERENCES

[DCOME83] D. COMER. The Computer Science Research Network CSnet: A History and Status Report. *Communications of the ACM 26*, 10 (October, 1983), 747–753.

[DCROC79] D.H. CROCKER, E.S. SZURKOWSKI, D.J. FARBER. An Internetwork Memo Distribution Facility — MMDF. Appearing in *Proceedings, Sixth Data Communications Symposium*, Asilomar, 1979, pp. 18–25.

[DCROC82] D.H. CROCKER. Standard for the Format of ARPA Internet Text Messages. Request for Comments 822. ARPA Internet Network Information Center (NIC), SRI International (August, 1982).

[DKING84] D.P. KINGSTON, III. MMDFII: A Technical Review. Appearing in *Proceedings Usenix Summer '84 Conference*, Salt Lake City, Utah, 1984, pp. 32–41.

[EALLM83] E. ALLMAN. SENDMAIL — An Internetwork Mail Router. Britton-Lee, Inc., Berkeley, California (July, 1983).

[IP] Internet Protocol. Request for Comments 791. Appearing in *Internet Protocol Transition Workbook*, ARPA Internet Network Information Center (NIC), SRI International, 1981.

[JREYN84] J.K. REYNOLDS. Post Office Protocol. Request for Comments 918. ARPA Internet Network Information Center (NIC), SRI International (October, 1984).

[MBUTL85] M. BUTLER, J.B. POSTEL, ET. AL. Post Office Protocol - Version 2. Request for Comments 937. ARPA Internet Network Information Center (NIC), SRI International (February, 1985).

[MROSE84A] M.T. ROSE. The Rand MH Message Handling System: The UCI BBoards Facility. Department of Computer and Information Sciences, University of Delaware (October, 1984).

[MROSE85B] M.T. ROSE, E.A. STEFFERUD. Proposed Standard for Message Encapsulation. Request for Comments 934. ARPA Internet Network Information Center (NIC), SRI International (January, 1985).

[SMTP] Simple Mail Transfer Protocol. Request for Comments 821. ARPA Internet Network Information Center (NIC), SRI International (August, 1982).

[TCP] Transmission Control Protocol. Request for Comments 793. Appearing in *Internet Protocol Transition Workbook*, ARPA Internet Network Information Center (NIC), SRI International, 1981.

Appendix A
MH Commands

MH is composed of several UNIX programs, which in theory are fairly simple and single-purposed. These commands are functionally grouped below:

Composing Mail

comp: compose a message
A program to originate a message. Usually, a special prompting editor front-end, *prompter*, is used to fill-in a composition template with the addressees of the message, subject, and so forth.

dist: redistribute a message to additional addresses
A program that re-enters a message previously received by the user into the message transport system. Only new addresses are added; the body of the message is not changed in any way.

forw: forward messages
A program that encapsulates one or more messages in a new message draft. In addition, the user may add initial and/or closing comments.

repl: reply to a message
A program that constructs a reply to a message using a reply template. The template mechanism has sufficient generality to permit the user to "program" the form of the reply draft based on the contents of the message being replied-to.

send: send a message
A program that posts a draft with the message transport system. The *send* program is usually invoked by one of the four preceding programs, and performs simple front-end pre-processing prior to invoking the *post* program. For example, if invoked in *push*'d mode, *send* will immediately relinquish control of the user's terminal and post the message in the background. If the posting fails, *send* will send back a failure notice to the user. If the user had *push*'d the sending of the draft, then by default the draft being sent is encapsulated in the failure notice. This permits easy *burst*'ing of the failure notice to retrieve the original draft. Otherwise, if the posting was successful, the draft is marked as having been sent.

whatnow: prompting front-end for send
A program which is called by *comp*, et. al., after the initial draft has been

generated. The MH user can specify a different *whatnow* program, which yields considerable extensibility.

whom: report to whom a message would go
A program which examines the addresses of the draft and expands all user-defined aliases contained therein. Optionally, *whom* may actually interact with the message transport system to determine the validity of the final addresses. This program is also usually invoked by *comp*, et. al.

Posting Mail

ali: list mail aliases
A simple front-end to the MH aliasing mechanism.

ap: parse addresses 822–style
A useful debugging tool for PostMasters who wish to examine how MH interprets an Internet address.

conflict: search for alias/password conflicts
Another program used by system administrators to check the consistency of MH alias files, and portions of the local message transport agent.

install-mh: initialize the MH environment
A program which is automatically executed the first time a user issues an MH command. This program performs once-only initialization of the user's MH environment.

mhmail: send or read mail
A simple program generally used by other programs to generate messages. The *mhmail* command is similar in purpose to the old *BellMail* program.

post: deliver a message
A complex MH back-end that interacts with the local message transport agent to enter messages through the posting slot. (See the description of *send* above).

Reading Mail

inc: incorporate new mail
A program that interacts with the local message transport agent to retrieve messages from the user's maildrop.

msgchk: check for waiting mail
A program which reports the status of mail waiting in the user's maildrop.

show: show (list) messages
A program which lists messages to its standard output (usually the user's terminal), possibly invoking another program to do the actual listing. Most users of MH have *show* automatically call the *mhl* program to format the

message. The *next* and *prev* programs are simply ''show␣next'' and ''show␣prev'', respectively.

mhl: produce formatted listings of MH messages
A program which displays a message as directed by a template. This permits the user to filter out uninteresting headers and re-arrange other headers to a particular preference. In addition to being invoked by *show*, the *mhl* program is optionally also invoked by *forw* to format each message being forwarded; invoked by *repl* to format the body of a message being replied-to, if that message is being included in the reply draft; and, invoked by *post* to format a message being sent as a blind-carbon-copy.

rmm: remove messages
A program that removes messages from an MH folder, optionally running a user-defined program instead of deleting them. If no program is given, the messages are "softly" removed, so they may possibly be recovered later.

scan: produce a one-line-per-message scan listing
A program that generates a scan listing for messages. Each line of the listing contains date, source, subject, and possibly the initial body of the message.

Folder Handling

folder: set/list current folder/message
A program used to list information concerning the current folder, or set the current folder and/or message.

folders: list all folders
A program to list information on all folders (actually, just a special case of the *folder* command). Since the MH folder structure may be recursive, the user can indicate that *folders* should recursively examine all folders.

refile: file message(s) in (an)other folder(s)
A program to move (or copy) messages from a source folder to one or more destination folders.

rmf: remove folder
A program that deletes a folder and all messages therein.

Message Selection

anno: annotate messages
A program to arbitrarily annotate messages. If the user so desires, after distributing, forwarding, or replying-to a message, MH will automatically attach an annotation to the original message indicating the date and addresses.

mark: mark messages
A program to manipulate user-defined sequences (lists of messages). Usually, *mark* is not employed directly by the MH user.

pick: select messages by content
A program to examine a list of messages and choose those which meet a particular selection criterion. The *pick* program is often used in UNIX back-quoted operations to pass message sequences to other MH commands.

sortm: sort messages
A program to sort a list of messages according to the date given in a particular field.

Distribution List Handling

bbc: check on BBoards
A front-end to run *msh* on a list of distribution lists which the user isn't current on.

bbl: manage a BBoard
A (depreciated) program used to manually manage the local archives of a distribution list. These functions (archiving, expunging) are performed automatically by MH.

burst: explode digests into messages
A program used to decapsulate messages from ARPA Internet digests. In addition, messages which have been encapsulated during forwarding (i.e.,' with *forw*) can also be decapsulated using *burst*.[10]

msh: MH shell (and BBoard reader)
A monolithic program used to implement MH commands on messages arranged in a single file (maildrop format). Useful since distribution lists are kept in this format to minimize consumption of system resources.

pack: compress a folder into a single file
A program which takes messages stored in MH format and places them in a single file (using the same format known by *msh*).

Interface to the UNIX File System

mhpath: print full pathnames of MH messages and folders
A program which maps MH-style names into the UNIX file naming convention.

[10] Similarly, blind-carbon-copies may be decapsulated, though only socially mature users should do so.

Appendix B
Distribution Mechanics

The mh.5 distribution is available in two forms:

1. Anonymous FTP
   If you can FTP to the ARPA Internet, use anonymous FTP to the ARPAnet host UDel-Huey [10.2.0.96] and retrieve the file portal/mh.5-tar. This is a *tar* image of size 2.1 MB (approximately).

2. 9-track tape, 1600 bpi, *tar* format
   Otherwise, you can send $50.00 to the address below. This covers the cost of a magtape, handling, and shipping. In addition, you'll get a laser-printed hard-copy of the MH documentation. The documentation includes installation guide, MH Tutorial, MH User's Manual, changes document (from mh.4 to mh.5), and BBoards Manual.

   If you go with this option, be sure to include your USPS address with your check. Checks should be made payable to

   Regents of the University of California

   It's also a good idea (though not mandatory) to send a computer mail message to `Bug-MH@UCI` when you send your check via USPS to ensure minimal turn-around time. The distribution address is:

   Support Group
   Attn: MH Distribution
   Department of Information and Computer Science
   University of California, Irvine
   Irvine, CA 92717

   714/856-6852

   Sadly, if you just want the hard-copies of the documentation, you still have to pay the $50.00. The *tar* image has the documentation source (the man is in ROFF format, but the rest are in TEX format).

In addition, there is some hope that mh.5, or a successor, might be found in a future 4.x Berkeley distribution.

Although MH is not "supported" per se, it does have a bug reporting address. Normally, the address Bug-MH@UCI is used to report bugs and bug fixes. There are however, two discussion groups which concern themselves with MH:

1. MH-Users@UCI
   A discussion group for the MH user community at large. Appropriate topics include: questions about how to use MH, tips on MH usage, and exchange of MH shell scripts. All requests to be added to or deleted from this list, along with problems, questions and suggestions, should be sent to MH-Users-Request@UCI.

2. MH-Workers@UCI
   A discussion group for MH maintainers and experts. Appropriate topics include: questions on how to configure MH, tips on MH configuration, exchange of MH bug reports (and fixes). All requests to be added to or deleted from this list, along with problems, questions and suggestions, should be sent to MH-Workers-Request@UCI.

The ''UCI'' host is also known as ''ucivax'', so a possible UUCP path might be ...!ucbvax!ucivax!bug-mh.

Updates to MH are published on the MH-Workers list in the form of context diffs, and the appropriate distribution images are updated as well.

In Search of a Better Malloc

David G. Korn
ulysses!dgk

Kiem-Phong Vo
ulysses!kpv

AT&T Bell Laboratories
Murray Hill NJ 07974

ABSTRACT

Dynamic memory allocation is a technique employed by programs to gradually use and release fractions of the computer's main or virtual memory resource. In the UNIX[1] operating system, dynamic memory allocation is provided to user programs via the library routines *malloc*, *free* and *realloc*. In this paper, we present a survey and comparative study of a number of algorithms implementing these routines. While no one algorithm is best in all respects, many show remarkable improvement in space and time measures over the ones standardly distributed with various versions of the operating system. We also present a technique to collect data from existing programs for post-analysis and a program to display such data in slow motion on a bitmap screen. This technique allows programmers with systems in which storage allocation is critical to choose the most suitable allocation algorithm. It has also uncovered bugs in the use of storage allocation in existing systems.

1. INTRODUCTION

Dynamic memory storage allocation is a technique commonly employed by programs to gradually use and release fractions of the computer's main or virtual memory resource. In the UNIX operating system, beside the low level system calls *brk* and *sbrk*, general mechanisms for dynamic storage allocation are standardly provided to user programs via the routines *malloc, free* and *realloc*. For brevity, we shall call these routines the *allocation routines*. The allocation routines manage a pool of contiguous memory storage blocks in the data section of the program in use. *Malloc* is called to request a block of storage of a specified length. *Free* is called to free a previously allocated block of storage. The freed storage is readied for future allocation requests. *Realloc* is called when the size of an active block of storage needs to be changed, either to get more space or to partially release some space at its tail end.

We have developed many programs using dynamic memory allocation. Performance profiling often revealed that the allocation routines took an increasingly significant fraction of the overall computing time. Wasteful space usage was also an occasional problem with these routines. In one case, using the standard first-fit routines (section 3.1.1), space fragmentation was so severe that a program ran out of memory and aborted after the data section had grown to several megabytes even though the required space was only about one hundred kilobytes. Perhaps because of the same experiences, many programmers have taken to using special allocation

1. UNIX is a trademark of AT&T Bell Laboratories.

techniques on top of the allocation routines to manage their space usage. A common practice was to use *malloc* to get large chunks of memory, then locally allocate from them for actual usage. We were, therefore, motivated to find out if there were better general purpose algorithms for dynamic memory allocation that would preclude the need for special case allocation techniques.

We have developed new allocation algorithms based on recent works on data structure techniques and ideas to improve the search time of the standard first-fit algorithm (section 3.1.1). By sending a request over the USENET network, we obtained a number of other general purpose implementations of the allocation routines. In addition, we obtained other versions from the system source directories. Having in our possession many different algorithms for dynamic memory allocation, a natural question that arose was how to evaluate these algorithms both in general usage and for particular applications. We have developed techniques for collecting memory usage data from existing programs with no change to the program source code, and to display such data dynamically on bitmap screens. Data collected in this fashion could be used to study typical space usage patterns of the involved programs and to see how different algorithms behaved with these patterns. Thus, using our techniques, a programmer could make intelligent choices as to which algorithms best suited his purposes.

This paper reports: a survey of the memory allocation algorithms that we have developed or obtained, a simulation study of the time and space behavior of the algorithms, and the techniques used to collect and display in slow motion storage usage data from existing programs. We found that the standard algorithms used in many flavors of the UNIX system (System V, 4.1BSD, and 4.2BSD) did not always behave well in measures of time, space or both. A surprising result was that, without careful consideration, an intuitively appealing improvement of the standard first-fit routine (section 3.1.1) could, in fact, decrease its efficiency.

2. MEMORY ALLOCATION ROUTINES

Each process running under the UNIX system consists of at least three sections: *text, stack,* and *data*. The text section, containing the program instructions, is fixed in size and generally unalterable during the life time of the process. The stack section, containing the initial environment variables and the call arguments to active routines as well as their automatic variables, is directly controlled by the operating system. The data section, containing initialized and uninitialized data, can be dynamically grown or shrunk by the process via the system calls *brk*[2] or *sbrk*. If the memory usage of a program is nearly stack-like, ie., the last requested block of memory is always the first to be freed, or if requested storage is seldom freed, *brk* and *sbrk* are quite adequate for memory management. For many programs, however, memory usage patterns are complex and more general mechanisms for memory management are required. In this case, the user level allocation routines, *malloc*[3], *free* and *realloc*, are standardly provided. These routines form a high level interface to the routines *brk* and *sbrk*.

As the allocation routines are used, the data section of a program grows and gradually fragments into blocks, each a contiguous section of storage. The storage of a block is properly aligned by the routines so that they can be used to store arbitrary elementary data, eg., *int*, *char* or *pointer*. At any given time, a block is in one of two states, *free* or *busy*. By agreement, the storage areas defined by busy blocks can be arbitrarily manipulated by the user program. The free blocks define areas of memory not directly used by the user program and available for future requests. Such free areas of memory, of course, can be and often are used by the allocation routines in their internal manipulations. Since the user program is allowed to refer to addresses in allocated

2. *brk*, section 2, UNIX System User's Manual.

3. *malloc*, section 3, UNIX System User's Manual.

blocks, the memory management model defined by the allocation routines does not allow relocation of data for compaction. The only compaction technique allowed is to merge adjacent free blocks.

Any general purpose implementation of the allocation routines must allow user programs to mix the system calls *sbrk* and *brk* with calls to the allocation routines. One reason for this requirement is that programs often make use of existing library routines and these routines may use the system calls directly to obtain storage. In such cases, the storage areas obtained by the system calls are treated by the allocation routines as permanently busy areas. We describe briefly below the functions of the allocation routines.

2.1 Malloc

The routine *malloc* is called as "*malloc(n)*" where *n* is the number of bytes requested. It searches the current set of blocks for a contiguous area of memory that is free and large enough. If such an area is not found, *malloc* calls the system routines *brk* or *sbrk* to increase the end of the data section and obtain new space. The new space is made into a new block. If the system call failed, *malloc* returns an error indicator. If the space found or created is larger than the request, depending on the implementation, it may be split into smaller free blocks, one of which is of the right size (typically, the lowest-address one). Finally, the address of the appropriate free area is returned to the user program.

2.2 Free

The routine *free* is called as "*free(p)*" where *p* is the address of a previously allocated block. It readies the space defined by the block for future allocations by marking it free and perhaps merging it with neighboring free blocks. In implementations where the free blocks are maintained separately, the new free block is inserted into the data structure maintaining the free blocks.

2.3 Realloc

The routine *realloc* is called as "*realloc(p,n)*" where *p* is the address of a previously allocated block and *n* is the new size in bytes. If *n* is less than the current size of the block defined by *p*, an appropriate amount of space is released from the tail end of *p*. If *n* is larger than the current size of *p*, more space is added to the tail end of *p* if possible. Otherwise, a new block is allocated, the data from *p* is copied to the new area, then *p* is freed. The address of *p* or the new data space is returned to the user program. Normally, *realloc* should be called only on busy blocks. However, in certain implementations, most notably, the standard first-fit algorithm (section 3.1.1) and its variants, *realloc* can be called on free blocks for space compaction. This feature is documented in the manual page of *malloc* in the UNIX System User's Manual but it is rarely used.

The function of *realloc* is a combination of that of *malloc* and *free*. *Realloc* exists for efficiency reasons, to compact unused storage space, and to save, if possible, data copying during a size change. In subsequent discussions, we shall often ignore *realloc*.

3. MEMORY ALLOCATION ALGORITHMS

The algorithms implementing the allocation routines are characterized by two factors, the policy used for space allocation, and the data structure used to manage the blocks of storage.

The space allocation policies fall into three classes: *first-fit, best-fit,* and *buddy methods*. In a *first-fit* policy, a memory request is satisfied from the first large enough free area found in some search order. The reader should note that this definition of the first-fit policy is different from the classical definition [Knu] in which blocks are searched in increasing order of their addresses. In a *best-fit* policy, a memory request is always satisfied from one of the smallest free blocks with sufficient size. In a *buddy method*, the task of managing blocks is reduced by limiting the set of sizes that can be allocated, for example, to powers of 2.

In an allocation algorithm, the space usage efficiency is strongly dependent on the space allocation policy used. The time efficiency of an algorithm is more dependent on the data structure used to manage the blocks of storage. The common data structures used are singly or doubly linked lists and binary search trees. In most cases, all blocks, free or busy, are linked in a singly linked list. To speed up search time, the free blocks are frequently maintained in a separate data structure.

In Figure 1, we list the algorithms that we studied in some detail. Each algorithm is assigned a one-character name for ease of reference in later discussion. For a simplistic measure of code complexity, in parentheses, we included the text sizes in bytes of compiled optimized object code on a VAX[4] 11/750 running the 4.2BSD UNIX system.

f: standard first-fit (*556*)
b: powers of two buddy method (*588*)
s: first-fit with singly linked free list (*700*)
d: first-fit with doubly linked free list (*2872*)
G: first-fit with self-adjusting free tree (*2692*)
S: better-fit with cartesian free tree (*2972*)
L: best-fit with balanced free tree (*3812*)
a: standard first-fit with adaptive exhausted search (*996*)
r: standard first-fit with restricted step search (*960*)
V: best-fit with self-adjusting free tree (*1948*)
K: multiple free list method (*1048*)

Figure 1. Studied Algorithms

The presented algorithms are suitable for general usage. They are also well-known or based on new ideas and presenting substantial improvements over existing ones. We omitted a few algorithms that do not satisfy these requirements, for example, one [Fow] that performs well only when few *free* calls are ever used and degrades quickly otherwise. Although for a large class of programs, this algorithm is entirely appropriate.

3.1 Algorithms That We Obtained

The algorithms described in this section were obtained from the system source directories or from users of the USENET network.

3.1.1 Standard First-Fit (f)

This algorithm is distributed with many versions of the UNIX system including the recent System V releases. It is *standard* in the sense that, unless otherwise directed, the link editor[5] automatically includes it for programs that use the allocation routines.

All blocks, busy or free, are linked in a singly linked circular list and ordered by addresses. Each block starts with a pointer pointing to the next adjacent block. Blocks are aligned on at least even addresses so that the low-order bit of the link field of a block is unused (it is always 0). This bit is then used to indicate the busy (bit on) or free (bit off) state of the block. A roving pointer is kept pointing at or near the most recently freed or allocated block.

An allocation request starts an *exhaustive* search from the block pointed to by the roving pointer. Adjacent free blocks are coalesced in this phase. If no suitable free block is found, a new free block is created at the end of the data section using the system call *sbrk*. If the found or created

4. VAX is a trademark of Digital Equipment Corporation.
5. *ld*, Section 1, UNIX System User's Manual.

free block is much larger than the requested size, an appropriate initial segment is made into a block to be allocated, and the remainder is made into a new free block. The allocated block is marked busy before being returned to the user program. The roving pointer is reset to the next block after the allocated block.

A free request marks the freed block free. The roving pointer is reset to point to this block. Since coalescing is performed only in the search phase of an allocation request, no coalescing is done at this time.

Each block is preceded by a one word[6] header containing the pointer linking it to the next adjacent block. Since *free* only resets a bit in the header of the freed block, this algorithm has the (dubious) characteristic that *free* and *realloc* can be called on previously freed blocks.

3.1.2 Powers of Two Buddy Method (b)

This algorithm was developed by C. Kingsley [Kin] and is distributed as the standard in the 4.2BSD UNIX system. A request size is rounded to a power of two (actually, a power of 2 minus the header size). Free blocks are kept in many singly linked lists, each for a particular size. No coalescing of free space is ever performed. The block header is one word in size.

3.1.3 First-Fit With Singly Linked Free List (s)

This algorithm was distributed with the 4.1BSD UNIX system as an alternative to the standard first-fit algorithm. The idea here is that the search for free space can be enhanced if instead of searching the entire pool of blocks, only free blocks are searched. Therefore, in addition to the singly linked list of all blocks, the free blocks are linked in a separate singly linked circular list and ordered by addresses. A roving pointer is kept pointing at the most recently freed or searched to free block. Free blocks are coalesced at *free* time.

Because of links in the free list, the header size is one word for busy block, and two words for free blocks. As with many subsequent algorithms, user's data is changed upon freeing so that *free* and *realloc* cannot be further called on previously freed block.

3.1.4 First-Fit With Doubly Linked Free List (d)

This algorithm is distributed with Release 2 of System V UNIX system as an alternative to the standard first-fit algorithm. Free blocks are linked in a doubly linked circular list with a roving pointer. New free blocks are inserted at the current position of the roving pointer (thus, they are essentially unsorted). In addition, very small blocks are not allocated directly. Instead, they are allocated from large holding blocks allocated via recursive *malloc* calls. Coalescing of adjacent free blocks is done in two stages. At *free* time, forward adjacent free blocks are merged. Other adjacent free blocks are merged during the search of *malloc*. The header size for a large free block is three words in this case.

3.1.5 First-Fit With Self-Adjusting Free Tree (G)

This algorithm was implemented by H. Gajewska [Gaj]. The free blocks are maintained in a self-adjusting binary search tree [ST] and ordered by their addresses. A self-adjusting tree is a binary search tree in which each insert/find/delete operation restructures the tree along the access path. The restructuring heuristics are only dependent on the local topological structure of the tree and do not require extra bits of information as in the case of balanced trees. The self-adjusting tree assures that, in amortized time [ST], the time spent to insert/find/delete blocks is of order O($n$log$n$) where n is the total number of such operations.

---

6. A word is a natural hardware size of pointer storage, usually 16, or 32 bits.

An allocation request starts a search from the root of the tree and stops when the first large enough free block is found. Coalescing of adjacent free blocks is done at free time. The header size of a free block is four words.

3.1.6 Better Fit With Cartesian Free Tree (S)

This algorithm was described by C.J. Stephenson [Ste], and implemented by C. Aoki and E. Adams [AA]. Pointers to free blocks are kept in a cartesian tree [Vui]. Let F be a node in the tree, representing a free block. We shall abuse notation and refer to F as the free block. The cartesian tree is maintained so that:

(i) address of any left descendant $<$ address of F $<$ address of any right descendant;

(ii) size of any descendant $<=$ size of F.

An allocation request starts a search for a free block from the root of the tree. At each decision point, the child with smaller size is chosen if it is large enough (hence, better fit). The search stops when both children are too small. The free block found is deleted from the tree. An appropriate amount of space is taken from it. The remainder if any is reinserted into the tree. A free request inserts the freed block into the free tree. Coalescing of adjacent free blocks are done at free time. Since the free tree is maintained separately from the blocks, each block, busy or free, has a one word header. However, there is correspondingly a tree node of size four words for each free block.

3.1.7 Best Fit With Balanced Free Tree (L)

This algorithm was implemented by B. Liblong [Lib]. The free blocks are kept in separate linked lists, each list containing all blocks of the same size. The different lists are kept in a balanced binary search tree [AHU] ordered by the block sizes associated with the lists. The balanced tree guarantees that, in the worst case, the time spent in insert/find/delete blocks is of order O($n$log$n$) where n is the total number of such operations. In addition, small blocks are kept separately in small block lists, each for a particular size.

An allocation request, if small, is satisfied from the small block lists. Otherwise, a search of the balanced tree is started for a list of elements whose associated size is smallest among all that are large enough. The first element of the found list is deleted from the list. An appropriate amount of space is taken from it. The remainder is inserted into a different list of free blocks. Adjacent free blocks are coalesced at free time. The header size for a big free block is five words.

3.2 Algorithms That We Developed

In addition to the above algorithms, we have developed four others based on a few reasonable heuristics and new data structure techniques.

Let the *end free block* be the free block that is immediately before the end of the data section (this block may be empty). In our experiment, we found that significant space saving results from not using the end free block unless it is desirable to do so. Following Stephenson [Ste] who coined the term *wilderness* for the area after the data section, we call this idea of not using the end free block the *wilderness preservation heuristic*. This heuristic was implemented with all of our algorithms.

3.2.1 Standard First-Fit With Adaptive Exhausted Search (a)

This algorithm is a variant of the standard first-fit algorithm, using the same data structure. Let A be the average size of a free block, and a a nonnegative real number. A can be easily estimated by keeping track of the total free space and the number of free blocks. For allocation, if the requested size is less than $(1+a)A$, an exhaustive search is performed. Otherwise, allocation is done from the end free block (which may have to be extended using *sbrk*). If an exhaustive search was performed and failed, the value of a is cut in half so that, next time around, the decision to search becomes more conservative. Note that if a was zero, the search

must succeed (prove!). On the other hand, if the search succeeded, the value of *a* is increased by a small constant to relax the search constraint.

3.2.2 Standard First-Fit With Restricted Step Search (r)

This is another variant of the standard first-fit algorithm. Let *A* be the average free block size, *n* the number of free blocks, *M* some fixed positive number larger than unity, and *f* some increasing function. For allocation, if the requested size is less than *A*, an exhaustive search for free space is performed. If the requested size is larger than *MA* then only the block pointed to by the roving pointer is checked before space is allocated from the end free block. For other sizes, a search for free space is started but it only goes *f(n)* steps before the last block is used. To prevent certain areas from not being reached, the roving pointer is made to continuously move around the circular list of blocks. That is, a call to *free* does not reset the roving pointer. In our experiment, we have obtained good results when *M* is *four* and *f* is the *square root* function.

3.2.3 Best Fit With Self-Adjusting Free Tree (V)

This algorithm is based on the best-fit allocation policy. The free blocks are kept in a self-adjusting binary search tree (section 3.1.5) and ordered by their sizes.

An allocation request is satisfied from one of the smallest free blocks that are large enough. Free blocks are coalesced at free time.

Each block, free or busy, has an one-word header containing the size of the block. Since the data section is a contiguous set of bytes, the size field also serves as a link to the next adjacent block. Block sizes are forced to be *0mod4* so that the least significant two bits of the size fields are free. A block is busy if its least significant bit is on; otherwise, it is free. The second least significant bit of the size field, if on, indicates that the previous adjacent block is free. When a block is free, its last word contains its own address. Thus, at a free request, adjacent free blocks of the to-be-freed block can be found and coalesced in constant time.

3.2.4 Multiple Free Lists Method (K)

Let *a, b, c, ...* be a finite increasing sequence of positive integers. The free blocks are partitioned in many doubly linked lists. The first list contains blocks with sizes between *a* and *b*. The second list contains blocks with sizes between *b* and *c* and so on. At an allocation request, let *s* be the requested size and *x, y* be two consecutive elements of the sequence so that *s* is between *x* and *y*. Then, the list associated with *x* is the only list with free blocks that may or may not fit the requested size. All lower lists contain blocks that cannot fit, and all higher lists contain blocks that must fit.

An allocation request first considers the initial element of the list *x* that contain elements that may or may not fit. If this element does not fit, the initial element of the next higher up non-empty list is used. The allocation request is then satisfied from the used element and the remaining part of the block is put on the appropriate free list. Free blocks are coalesced at free request time.

The sequence *a, b, ...* used in the algorithm is arbitrary and can be customized by user programs. In our experiment, the subsequence *F(6), ... F(21)* of the Fibonacci sequence is used. The Fibonacci sequence is defined by the following recursive equations:

```
F(1) = F(2) = 2
F(n) = F(n-1) + F(n-2) for n > 2
```

The structure of a block here is similar to that in the above best-fit with self-adjusting tree algorithm except that the size field is replaced by an address field where the address points to the next adjacent block. In this case, blocks are aligned on addresses *0mod4* so that the last two bits of address fields are free.

4. SIMULATION EXPERIMENT

Theoretical and experimental studies of the space performances of different allocation strategies in particular, best-fit and classical first-fit, have been done under different statistical distributions of usage patterns [Cam] [Knu] [MPS] [Sho] [Ste]. The time performances of different allocation algorithms have not received as much attention. This is because the time measures are highly dependent on the support data structures and the quality of programming in the implementations. In this section, we report results from a simple simulation study of the time and space performances of the various implementations in our possession.

4.1 Simulation Technique

The simulation technique used in our study is similar to that described in [Knu, p446]. The heart of the simulation program is presented below in the C language [KR]. We should point out quickly that the code presented only shows the spirit of what was done. In the actual experiment, since the same random data was used with all different algorithms in several runs, the random data was generated once by a separate program. The data was then read in by the simulation program before the loop was entered.

```
for(t = 0; t < MaxTime; ++t)
        {
        s = randsize(MeanSize);
        l = randlife(MeanLife);
        reserve(s,t+l);
        freeall(t);
        }
```

Figure 2. The simulation algorithm

A simulation session runs *MaxTime* time units. At each time unit t, two independently random numbers s and l are set. The mean values of these random numbers are *MeanSize* and *MeanLife* respectively. A new block with size s is reserved and scheduled to be released at time $t+l$, ie, l steps in the future. Then, the routine *freeall* releases all allocated blocks scheduled to be freed at time t.

In each run, the simulator records four quantities, M, the maximum allocated space at any given instance in time, A, the maximum length added to the data section during the simulation, and U and S the total cpu and system times used in the *for* loop. The quantity $(A-M)/A$ measures the wastage in space management of the algorithm in use and the quantity $U+S$ measures the total time used.

The conditions of the experiment were as follows:

- The experiment was carried out at night on an unloaded VAX 11/750 running the 4.2BSD UNIX system.
- To reduce variation, for each algorithm and each of the statistics (M, A, U and S), three data points were collected for each triple of *(MaxTime,MeanSize,MeanLife)*. Then, the average value of the data points was used as the final data for the triple.
- *MaxTime*, *MeanSize* and *MeanLife* were varied respectively between 2500 to 25000 in increments of 2500, 250 to 750 in increments of 125, and 125 to 1250 in increments of 125.
- The distributions of the size and life time of allocated blocks were independent and uniform over the intervals *[1,2MeanSize]* and *[1,2MeanLife]* respectively.

4.2 Simulation Results

In this section, we present data on the space usage and time performances of the various algorithms. First, we show that the space usage of the algorithms reached a steady state well within the bounds of our experiment. Then we show the comparative performances of the algorithms when the steady state has been reached.

Figure 3 shows the space wastage varying with *MaxTime* when *MeanSize* and *MeanLife* were fixed at 500. As shown, the space wastage reached a steady state well within the maximum value 25000 of *MaxTime*.

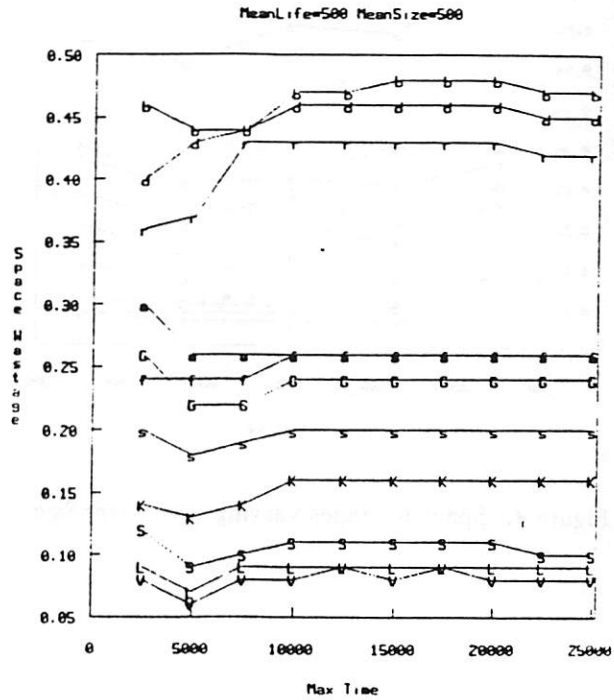


Figure 3. Space wastages varying with *MaxTime*

Figure 4 shows the space wastage varying with *MeanSize* when *MeanLife* and *MaxTime* were fixed at 500 and 25000 respectively.

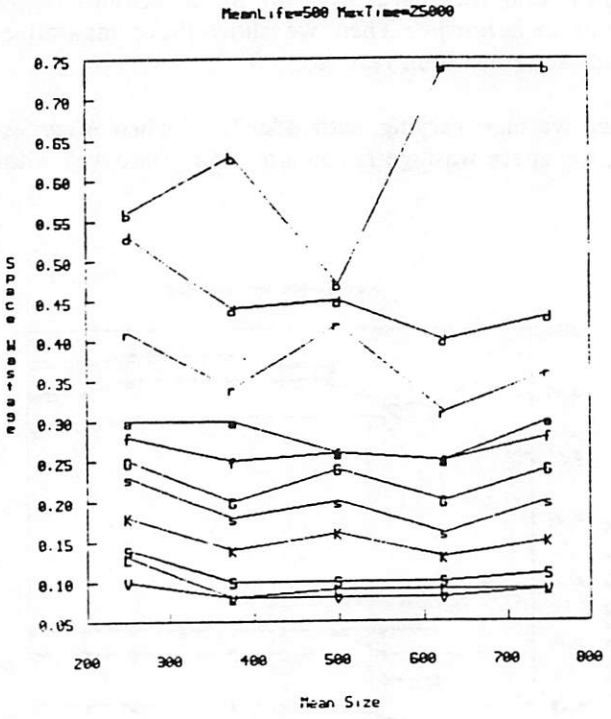


Figure 4. Space wastages varying with *MeanSize*

Figure 5 shows the space wastage of the various algorithms relative to the mean life time of allocated blocks when *MaxTime* is fixed at 25000 and *MeanSize* is fixed at 500.

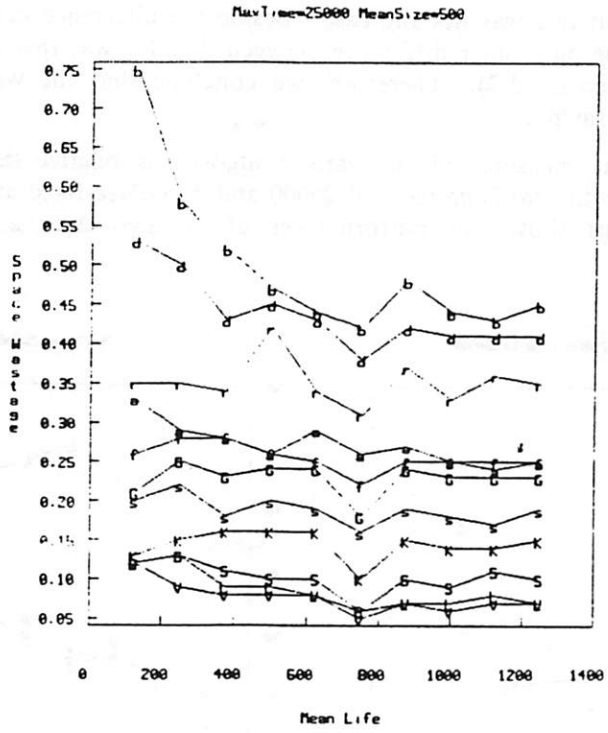


Figure 5. Space Wastage varying with *MeanLife*

Figures 4 and 5 show that the buddy method performed worst, wasting between 45 to 75 percent of space. As expected, the wastage decreased as the life time of allocated pieces increased since there are fewer free blocks. Generally, the best-fit algorithms *L* and *V* performed best, wasting only between 5 to 12 percent of total space used. The reader should also be aware of the experimental results of [Sho] in which the best-fit policy was compared with the classical first-fit policy. The experiment in [Sho] showed that best-fit outperformed classical first-fit when requests for large blocks approaching a good fraction of total memory were infrequent. On the other hand, when such large requests were often, classical first-fit outperformed best-fit. Under the conditions of our experiment (relatively small block sizes comparing to the available storage), our results concurred with the conclusions of [Sho] that best-fit should perform well. Given that machines with primary memory storage exceeding the megabyte limit have been becoming the norm, our underlying assumption that request sizes were much smaller than memory size was realistic.

Following the best-fit algorithms were the better-fit algorithm *S* and the multiple free list algorithm *K* whose wastages varied between 10 to 15 percents. The good performance of the *K* algorithm could be attributed to its partitioning allocation strategy which was a tighter-fit strategy.

Next in space performance was the first-fit family of algorithms. The relatively good performance of algorithm *s* and *G* could be attributed to their strategy of coalescing adjacent free space at free time. The poor behavior of *r* can be explained by the fact that some of the free space in the arena might not have been found because of the restricted number of steps.

Algorithm *d* did not do well because free blocks were not ordered by addresses so that during the search phase, certain adjacent free blocks may not be coalesced, resulting in more fragmentation.

Of interest is the comparative space performance of the standard first-fit algorithm *f* and its adaptive search variant *a*. In algorithm *a*, there were times in which an exhaustive search for free space was skipped while it would be done in *f*. Therefore, one should expect *f* to do better than *a* in space utilization, but this was not the case. Beside the difference in when to exhaustively search for free space, the only other difference between *f* and *a* was that *a* used the wilderness preservation heuristic (section 3.2). Therefore, we conclude that the wilderness preservation heuristic helped in space saving.

Figure 6 shows the time measures of the various algorithms relative the mean life time of allocated blocks, again with *MaxTime* fixed at 25000 and *MeanSize* fixed at 500. The figure has two parts, the first part shows the performances of all algorithms and the second omits algorithms *f* and *s*.

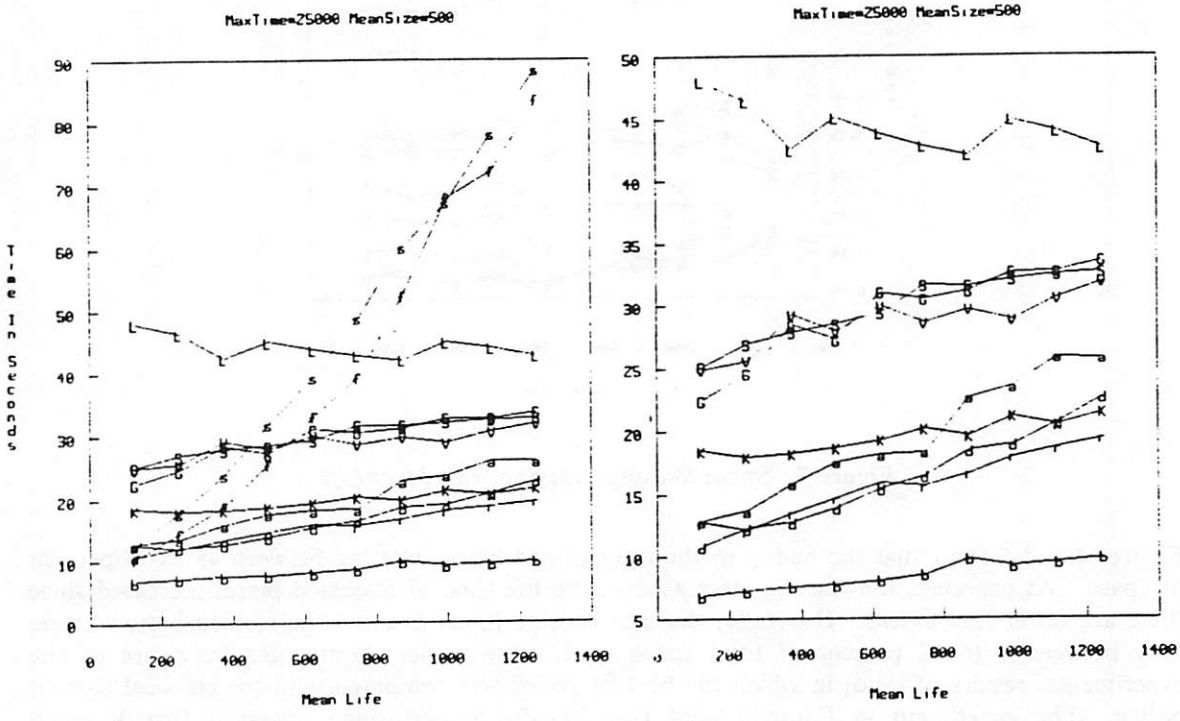


Figure 6. Time Measures Varying with *MeanLife* - 1

Here the time performance of the buddy algorithm *b* was best since its computations were extremely simple. Of the tree based algorithms, the performance of the self-adjusting tree algorithms *G* and *V* and the cartesian tree algorithm *S* were close together. This is to be expected since at equilibrium, the free trees tend to be balanced. In light of this, it was surprising was that the balanced tree algorithm *L* performed much worse than the others. This could be attributed to the fact that much more information was needed to maintain balanced trees than to maintain the other types of trees.

The rapid rising behavior of the standard first-fit algorithm *f* was directly attributed to the fact that it exhaustively searched for free space in the entire linked list of free and busy blocks. What was surprising was that algorithm *s* which maintained the free blocks in a separate list performed even worse than *f*. A moment of reflection, however, showed that even though *s* could search

for free space faster than *f*, it also spent much more time to maintain the *singly linked* free list. This was because in order to coalesce a free block with its preceding adjacent free block, the algorithm had to do a linear search of the free list for the preceding element. Thus, if the allocation pattern was such that there were frequent *free* calls compared to the total number of allocations, *s* would perform worse than *f*. This hypothesis was further supported in the next figure in which the *MaxTime* parameter was cut to 15000 to reduce the frequency of *free* calls relative to *malloc* calls. In that case, algorithm *s* performed better than algorithm *f*.

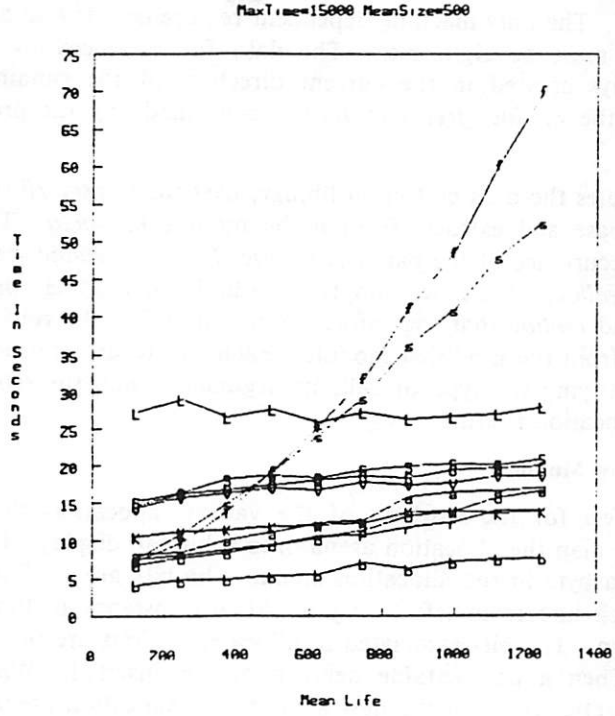


Figure 7. Time Measures Varying with *MeanLife* - 2

5. COLLECTING AND DISPLAYING DATA FROM EXISTING PROGRAMS

We were interested in methods for collecting data from existing programs and displaying them dynamically. Such methods are useful since they can be used by programmers to find out which allocation algorithms best suit the storage usage patterns of their programs. In this section, we describe the data collection technique. Then, we describe a program to display in slow motion the collected data on a bitmap screen. Using the program, we were able to identify the cause of a worst case fragmentation in the standard first-fit algorithm.

5.1 Data Collection Technique

Since many existing programs are not accessible to us, we have to develop a data collection technique that can be used by other users to collect storage usage data. To be successful, the technique must require as little understanding of the subject programs as possible. Ideally, we would like to be able to gather the data using only a modified version of the binary executable code [7] of any program. However, there does not seem to be any easy way to do this. Further,

any method that manipulates the binary executable code is highly dependent on the hardware and the version of the resident UNIX operating system.

The method that we chose is a compromise that does not require any understanding of the source code of a program but it requires a partial relinking. We supply a software to generate a *data collection library* of routines interfacing the allocation routines of the resident operating system. Users only need to relink the object modules (*.o* modules) of their programs with the routines of the data collection library. The interface routines, in addition to satisfying storage requests, generate a data file containing the storage usage pattern of the running program. The representation of the generated data is nearly machine independent. In particular, byte ordering is properly taken care of. The only machine dependent requirement is that a *long* integer must be four bytes and a byte must be eight bits. The data file produced by the library is called *malloc.data*. It is always created in the current directory of the running process. The file contains a trace of all the *malloc*, *free* and *realloc* calls made by the program or the library routines that it uses.

The software that generates the data collection library, uses the library */lib/libc.a* of the resident operating system as a base and extracts from it the module *malloc.o*. This binary module is modified so that each occurrence of the patterns *malloc*, *free* and *realloc* are changed respectively to *Malloc*, *Free* and *Realloc*. Then, we supply an additional module containing the interface routines *malloc*, *free* and *realloc* that log information concerning the routine calls then call the corresponding routines from the modified module. Each call to an interface routine generates a twelve byte record identifying the type of call, its arguments, and the returned value from the actual corresponding allocation routine.

5.2 Displaying Data In Slow Motion

To help get a better feel for the working of the various allocation algorithms, we wrote a program to dynamically map the allocation arena onto a bitmap display. Each bit in the bitmap display corresponds to a byte in the allocation arena. The bits are set line by line from top to bottom, and within each line from left to right. At any instance in time, a snapshot of the allocation arena is shown. The bits associated to allocated regions are turned on while the other bits are turned off. Then a user-settable delay period is inserted. When the delay time is finished, the display is refreshed with the new state of the allocation arena. Therefore, in slow motion, we can observe the way a particular algorithm behaves.

The following example shows how the display program works. Figure 8 shows a fragment of a C program using the allocation routines. The integer variable *size* contains the current size of the storage block whose address is stored in *large*. The symbols *DELTA* and *EPSILON* are predefined constants with some small values.

7. *a.out*, Section 5, UNIX System User's Manual.

```
size = DELTA;
large = malloc(size);
small = malloc(EPSILON);
while(1)
            {
            free(large);
            size += DELTA;
            large = malloc(size);
            small = malloc(EPSILON);
            }
```

Figure 8. An example of dynamic memory allocation

Figure 9 shows the allocation arena when the standard first-fit algorithm *f* was used. The terrible fragmentation shown in Figure 9 was traced easily to the way in which the standard first-fit algorithm handled the roving pointer and the free block at the beginning of wilderness. When *size* was more than a kilobyte, the *malloc(size)* call often caused a call to the system call *sbrk* to increase the data section. When this happened, a large enough number of kilobytes were obtained to satisfy the request. The roving pointer was then set to point to the excess amount of the newly obtained memory, causing the next call *malloc(EPSILON)* to block the end of the large block. Therefore, at the next iteration, even though the *large* block was freed, there was not enough memory to satisfy its next allocation. This problem can be cured by using the wilderness preservation heuristic.

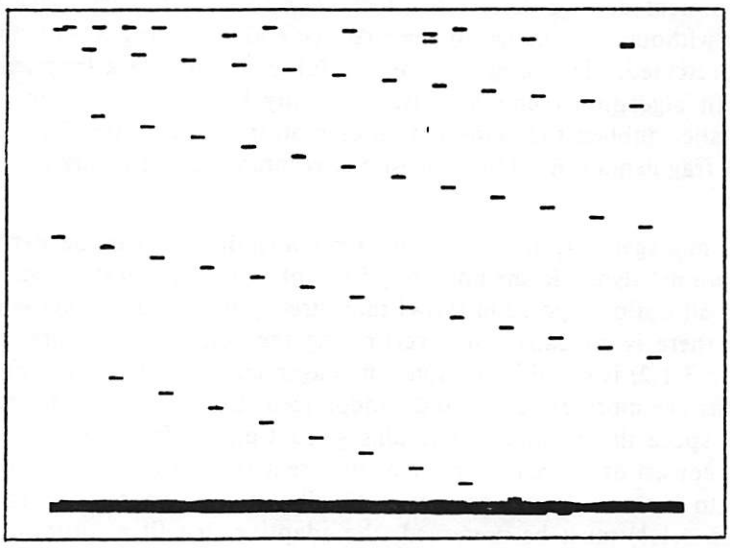


Figure 9. A snapshot of the arena using algorithm *f*

Figure 10 shows a snapshot of the allocation arena when the best-fit with self-adjusting tree algorithm was used. In this case, the space usage was much more reasonable.

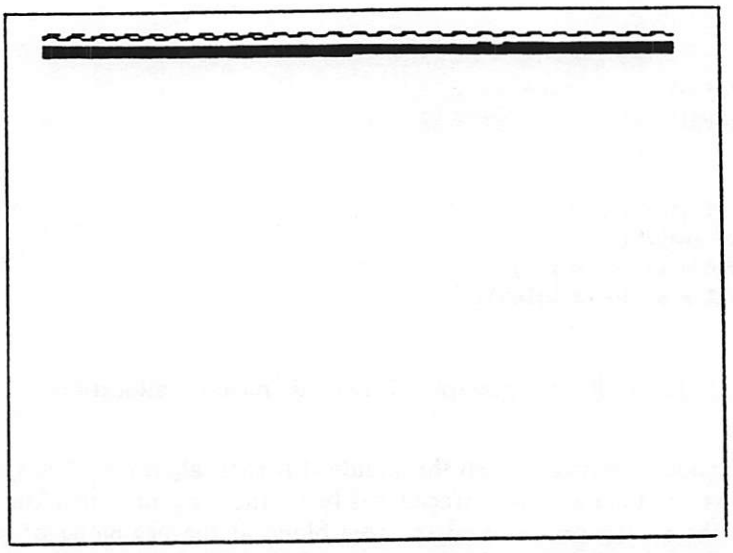


Figure 10. A snapshot of the arena using algorithm *V*

6. DISCUSSIONS

In this report, we have given a survey of eleven algorithms implementing the UNIX system allocation routines. A simulation study of the algorithms was performed and the relative merits of the algorithms presented. We described a technique to collect space usage pattern data from existing programs without any change to their source code. A program to display such data in slow motion was presented. The program was helpful in identifying a fragmentation anomaly in the standard first-fit algorithm commonly used in many UNIX systems. In our work, we also found that a heuristic, dubbed the wilderness preservation heuristic (section 3.2), helps reducing space wastage and fragmentation. This heuristic is recommended for any implementation of the allocation routines.

For general space management, the following recommendations can be derived based on the knowledge of programs' dynamic memory requirements. If a program never or seldomly frees allocated space, an allocation algorithm [Fow] that directly interfaces the system call *sbrk* and *brk* is appropriate. If there is no danger of overflowing the computer memory, the buddy method algorithm *b* (section 3.1.2) is suitable for space management since it is fast. For programs whose space usage patterns are more extensive and randomized, the best-fit algorithm *V* (section 3.2.3) is good since it is space thrifty and still retains good time performance. For programs whose allocation patterns consist of a small number of different sizes, algorithm *K* (section 3.2.4) is best since it is tunable to such size usage patterns. Finally, if the semantics of the standard first-fit algorithm *f* (section 3.1.1) must be preserved, the adaptive first-fit algorithm *a* (section 3.2.1) is an appropriate replacement since it is both faster and more space economical than *f*.

There are, of course, a variety of programs whose usage patterns do not yield obvious choices of algorithms. In such cases, the data collection and display techniques that we developed can be used to collect run-time data for post-analysis and to determine which allocation algorithms are best suited for the frequent space usage patterns. We are considering automating this process.

There are a number of potentially important issues ignored in our study. We did not attempt to answer which algorithm performs well in a virtual memory environment. In this case, the number of pages accessed to allocate a block and the total number of pages used are the primary concerns. Therefore, algorithms that focus on page alignment may be more important than

internal fragmentation. We have not developed adequate tools to measure the performances of such algorithms nor did we have any of these algorithms in our collection. On the other hand, it should also be noted that if all allocated space are accessed frequently by user programs, intuitively, algorithms that reduce fragmentation still perform best because there are fewer memory pages to be accessed.

The issue of robustness with respect to asynchronous events was also omitted. As far as we can determine, none of the algorithms that we tested are completely reliable. If a signal[8] handler occurs while in the midst of a free or an allocation, and it makes an allocation or a free request as part of its actions or if it does a *longjmp*[9] then all the algorithms we tested can fail because the data structures maintaining blocks may lose their integrity. Note that the allocation routines can be programmed to delay signals but the cost of doing this is often prohibitive. Some algorithms are better than others with respect to such asynchronous events. Algorithms such as the standard first-fit *f* or the buddy method *b* that maintain simple structures of blocks are more stable than algorithms that use complicated data structures to manage free space. The tree and doubly linked list algorithms can have their data structures so mangled that later accesses to them can cause memory faults. If asynchronous event handling of the types described is necessary, the adaptive search variant *a* of the standard first-fit algorithm *f* is an attractive choice.

ACKNOWLEDGEMENTS

We thank all users of USENET who responded to our query, especially, Hania Gajewska, Steve Muchnick and Alan S. Watt who shared information and thoughts on the subject.

REFERENCES

[AA] C. Aoki and E. Adams, Private Communication, 1984.

[AHU] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[Cam] J.A. Campbell, *A Note on an Optimal-Fit Method for Dynamic Allocation of Storage*, The Comp. J., v.14, n.1, pp.7-79, 1971.

[Fow] G. Fowler, Private Communication, 1984.

[Gaj] H. Gajewska, Private Communication, 1984.

[Kin] C. Kingsley, 4.2BSD UNIX System.

[Knu] D.E. Knuth, *The Art of Computer Programming, Vol I, Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1968.

[KR] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[Lib] B. Liblong, Private Communication, 1984.

8. *signal*, Section 2, UNIX System User's Manual.
9. *setjmp*, Section 3, UNIX System User's Manual.

[MPS] B.H. Margolin, R.P. Parmelee and M. Schatzoff, *Analysis of Free Storage Algorithms*, IBM Sys. J., v.10, n.4, 1971.

[PN] J.L. Peterson and T.A. Norman, *Buddy Systems*, CACM, v.20, n.6, pp.421, 1977.

[Sho] J.E. Shore, *On the External Storage Fragmentation Produced by First-Fit and Best-Fit Allocation Strategies*, CACM, v.18, n.8, pp.433-440, 1975.

[Ste] C.J. Stephenson, *Fast Fits: New Methods for Dynamic Storage Allocation*, Ninth ACM Symposium on Operating Systems, 1983, SIGOPS Operating Systems Review, v.17, n.5.

[ST] D. Sleator and R.E. Tarjan, *Self-Adjusting Binary Search Trees*, Private Communication, 1984.

[Vui] J. Vuillemin, *A Unifying Look at Data Structures*, CACM, v.23, n.4, pp.229, 1980.

The Impact of Buffer Management on Networking Software Performance in Berkeley UNIX™ 4.2BSD: A Case Study.

Luis Felipe Cabrera ‡
Michael J. Karels
David Mosher †

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720, USA

Abstract

Berkeley UNIX 4.2BSD is an operating system that provides easy networking among 4.2BSD installations and others supporting DOD's Internet protocols. Moreover, it also offers alternative ways for processes to communicate with each other both within and across machine boundaries. Processes need not have a common ancestor to communicate and they may do so using different addressing families and styles of communication. In addition, several protocol families may be supported simultaneously.

In this paper we present a detailed timing analysis of the dynamic behavior of the TCP/IP and the UDP/IP network communication protocols' current implementation in Berkeley UNIX 4.2BSD. These measurements show the effect that kernel buffer management has on the network software performance. We discuss issues and tradeoffs involved when implementing network communication mechanisms for multiple-protocol systems. We highlight the intricate interrelationships arising from the simultaneous coexistence of different buffering policies within a system.

This study also sheds light on the inefficiencies encountered when software and hardware perform the same actions on data, e.g., checksums.

Index Terms: Berkeley UNIX, 4.2BSD, benchmarking, interprocess communication, datagram, virtual circuit, TCP protocol, UDP protocol, IP protocol, Ethernet, artificial workload, dynamic program profile.

1. Introduction

Berkeley UNIX 4.2BSD is an operating system which provides alternative ways for user processes to communicate with each other [9-10, 19, 23]. User processes may choose an inter-machine communication medium, protocols [15-17], addressing families, and styles of communication. In particular, user processes may use datagram or stream communication. In Berkeley UNIX 4.2BSD two processes wishing to communicate need not have a common ancestor nor reside in the same host. In this paper we present a study of the dynamic behavior of TCP/IP and UDP/IP in which the impact that kernel buffering has on network software performance is highlighted. We have studied the implementations which existed at Berkeley during the summer

---

‡ On leave from the Departamento de Ciencia de la Computación of the Escuela de Ingeniería of the Pontificia Universidad Católica de Chile.

† Author's present address: Pyramid Technologies, 1295 Charleston Road, Mountain View, CA 94039.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD), monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Goverment.

of 1984, nine months after 4.2BSD release, in Ethernet based environments [13-14]. Hereafter, we shall refer to this version of Berkeley UNIX 4.2BSD as the 'current' system. (The system we measured already differed in many aspects with the released 4.2BSD Berkeley UNIX.)

A secondary aspect we wanted to observe was the possible effect that the reliability built into the TCP protocol [17, 14] has on the user process-perceived network latency and overall network performance. Two questions we are concerned with are: What are the tradeoffs of using TCP compared to UDP if the underlying physical network does not lose packets? Given that some Ethernet interface hardware provides CRC checksums, what is this cost for user applications when using TCP?

The rest of this paper is subdivided as follows. In Section 2 we present the basic measurement assumptions. Section 3 has a discussion of the experimental environment used in this study. Section 4 contains a detailed dynamic timing analysis of the current TCP/IP and UDP/IP implementations. Finally, Section 5 consists of our conclusions.

2. Timing From User Process Space

Even though many studies have evaluated and modelled the performance of Ethernets under various conditions [1-4, 6-7, 18, 20-21], measurements performed in most of those studies have only determined the throughputs which can be achieved between hardware interfaces, and not, say, from the user level. Indeed only [2, 7, 21-22] have measurements of the performance a software system may expect. However, in [21-22] those measurements do not refer to communications between user processes but between operating system processes. In [8] we find an analysis includes modelling the behavior of Ethernets under different parametric load conditions, in the context of file servers.

In this paper we study the network software performance perceived by user processes. It should be emphasized that all measurements were made from user space, and thus they include all the overhead caused by the protocol implementations and by the operating system. Better understanding of how our local area network implementations perform makes it easier to design distributed applications. It also provides a better knowledge of the current limitations of these distributed applications, and where the improvements may come from.

3. The Experimental Environment

This paper describes a series of tests, performed at UC Berkeley, designed to determine the dynamic performance properties of the TCP/IP and UDP/IP Ethernet-based IPC mechanisms under Berkeley UNIX 4.2BSD. Given that IP, as currently implemented, cannot be accessed directly by user processes (but can by privileged processes), we shall not report explicitly about it in this paper. It should be clear, however, that since UDP and TCP use IP, the performance of its implementation affects user process applications.

For our study we used the kernel configured for profiling and executed ad hoc routines to stress desired aspects of the network subsystem. The assessment was then done through the use of the commands *kgmon* and *gprof* [5, 11]. *gprof* profiling is known to add 15 to 20% cpu overhead to the profiled code [11]. This should be kept in mind while reading the timings in Section 4. A test program which sends a fixed amount of data a predetermined number of times was designed for each protocol. The kernel monitoring facility was enabled during the execution of the test program. All tests were run in single user mode to avoid interferences. The profiles were determined later from the data collected by the monitor [5].

3.1. The Sizes of Messages and the Repetition Count

The six user message sizes used for the network tests were: 1, 112, 113, 1023, 1024, and 1025 bytes. This set of message sizes was chosen to stress border conditions in the buffer management schemes provided by the protocol implementations. Some of them also represent traffic resulting from common network operations, as is the case with the 112 byte size and the 1024 byte size. Most system utilities use, for example, the 1024 byte size as it represents one

logical page of data in Berkeley UNIX 4.2BSD. Indeed the networking software has been optimized for the 1024 byte size.

The networking software manages its own address space with two sizes of buffers: 128 bytes and 1024 bytes. Mbufs are 128 byte buffers where 8 bytes are used for link pointers, 4 bytes for the data offset, 2 bytes for the size, and 2 bytes for the type. Mbufs may contain up to 112 bytes of data, or they may point to an associated 1024 byte data area which always contains exactly 1024 bytes of data. User supplied data is copied into a chain of mbufs with 1024 byte buffers for each segment that is at least a page in size, and any remainder (or all if the segment is less than 1024 bytes) is copied into mbufs containing up to 112 bytes each. Internal copy operations involve physical copying of the data segments inside of mbufs. Mbufs which point to 1024 bytes of data are copied by augmenting an associated reference count. Internal copy operations are required by the retransmission algorithm with TCP and for messages to be assembled into one contiguous buffer after being given to the network driver. In Section 4 the effect that saving one of these copy operations has on network performance will become apparent. IP protocol message fragmentation was not assessed in this study.

It should be remarked that, for every message to be sent, the network software always prepends a 128 byte mbuf for the 40 byte header that is required by the TCP/IP and UDP/IP protocols. Messages with 1024 byte data segments may be transmitted by the link layer using a trailer protocol [9, 12], which allows the TCP/IP or UDP/IP header to be moved to the end of the data area in order to page-align the data area for both the transmitting and receiving hosts. This optimization, which avoids message reassembly costs at the receiver end of a transmission, can only be done if the headers of messages have constant length.

Each network test consisted of a user process sending a fixed size message a predetermined number of times. These tests were run between two VAX 11/750's over a private 3 megabit/second ether. One processor was used to profile the kernel while the test program ran; the other otherwise idle processor just sinks the data from the test programs. We call 'repetition count' the number of times each packet was sent. The higher the repetition count, the larger the degree of accuracy we could obtain from our profiling tools. To obtain values accurate to one millisecond a repetition count of 10,000 was necessary. The same repetition count was used in each of our runs.

4. Assessment of Protocol Implementation

In this section we present the study of the implementations of TCP/IP and UDP/IP in detail. The primary goal of this study is to understand the specific costs of using TCP/IP and UDP/IP as currently implemented. Secondarily, this study has pointed out unexpected performance penalties for certain message sizes.

Two similar test programs were designed, one for each protocol, which sent a specified number of messages of a specified length to a predetermined host. In each run, from the profiled kernel we obtained the execution history of sending a message with the specified amount of data. The kernel profiling facilities were enabled only during the actual running of each test program. The test programs were run in single user mode to avoid interferences as much as possible. The test programs have been included in Appendix A.

It must be remarked, however, that lacking a hardware monitor to measure overall ether message size distribution we did not have an absolute way of judging the effectiveness of the current protocol implementations for the user process applications in general. What we do have, however, is an excellent breakdown of kernel time spent while sending messages of the chosen sizes, and a detailed knowledge of what would happen if, say, a user space application sent messages of any given size.

The raw data from these profiles appears in the following three subsections. The values in Tables 1 and 2 represent the number of seconds spent processing in each routine during the 10,000 transmissions. Because these values were obtained from a single run for each message size, they are mostly intended to show the relative ordering of the routines with respect to

processor utilization, and to show gross changes in the magnitude of time utilization of each routine. Obtaining absolute timings would require further data stability analysis. (Some of the message sizes were run more than once on different days. No significant timing differences were observed.) In Tables 1 and 2 we have highlighted in boldface those routines which exhibit larger timing variations as a function of the amount of data to be processed. We have not tried to factor out the 'Heisenberg' effect of *gprof*'s overhead.

For both protocols, the buffer scheme used in the implementation appears to have an overwhelming effect on performance. Since UDP/IP sends data atomically, and only limits a message's maximum size, this protocol is not so sensitive to varying data sizes. The drastic increases in overhead in the routines shown in bold appear to be due to the data buffer management scheme chosen. On the other hand, TCP/IP, with its windows and data streaming, is sensitive to the amount of data presented. Thus, in addition to the increased overhead seen in the routines which deal with data within the buffer management system, the actual protocol implementation overhead appears to increase noticeably because of the varying amounts of data presented.

A word of caution. Our analysis of TCP/IP was done based on a user process which sent data using *write*, while that of UDP/IP used *sendto*. These calls have, for example, different number of parameters and thus their overheads have to be compared with care.

4.1. TCP/IP

Table 1 presents the time spent in a selected group of routines that are called to process a TCP/IP transmission with a specific amount of data. These values were obtained directly from the *gprof* output.

The calling hierarchy for sending data via TCP/IP starts with a system call, *syscall*, to a generic *write* operation. *write* calls *rwuio* to set up transfer data structures. In turn, *rwuio* calls the specific routine which can perform the necessary operation for the type of object, in this case *soo_rw*. *soo_rw* calls the appropriate internal routine that implements the original request to 'send data', *sosend*. *sosend* is responsible for allocating buffers and copying the data from the user space via *uiomove*, which calls *Copyin* to do the actual copying. *sosend* first determines the amount of buffer space available for this specific socket, and then copies the minimum of the buffer space available or the amount of data to be sent, whichever is smaller, into mbufs. These mbuffs are then passed to the appropriate protocol, in this case *tcp_usrreq*. *tcp_usrreq* queues the data buffers for this TCP connection with *sbappend* and then switches immediately to *tcp_output*, the output sequencer for the TCP protocol. Based on the windowing policies, an amount of data to be sent is selected. This data is copied from the TCP output queue by *m_copy*. Data and header are checksummed in *tcp_cksum*, and then passed to the IP level, *ip_output*. The additional IP header information is checksummed in *in_cksum*. Finally, the message is queued and possibly sent to the specific network interface for transmission. In this study, the network interface is represented by the two functions *en_output* and *en_start*. Before such a transmission can happen, the buffered data must be copied and mapped into a single contiguous memory space; this is done in *if_wubaput*.

From the row entries in Table 1, we can see that, of the 21 different routines listed for TCP/IP, 12 present processing costs which vary significantly with the amount of data sent. The processing time of the other 9 routines remains practically constant. The five calls which show a larger variation in the vicinity of the 1024 bytes region are *sosend*, *uiomove*, *m_copy*, *sbappend*, and *if_wubaput*. All are associated with buffer management. The four calls which have a larger impact in the processing of messages are *sosend*, *if_wubaput*, *tcp_cksum*, and *m_copy*. *tcp_xoutput*, which is a copy of *tcp_output*, is only called from *tcp_input*. This allowed us to observe in isolation the cost of the flow control mechanism and of packet acknowledgement. Clearly, the greatest impact comes from those routines do copying of data within the interfaces. In the 1024 case, checksumming and servicing acknowledgements and window updates through *tcp_input* and *tcp_xoutput* are the most expensive tasks.

TCP/IP Routines and System Calls	Message Size in Bytes					
	1	112	113	1023	1024	1025
syscall	2.91	2.79	2.76	3.02	3.01	3.12
write	0.72	0.81	0.81	0.78	0.84	0.90
rwuio	1.57	1.67	1.82	1.77	1.84	2.23
soo_rw	0.92	0.83	0.85	0.82	0.77	0.84
sosend	**3.89**	**4.76**	**6.24**	**15.53**	**5.53**	**16.52**
uiomove	**0.99**	**1.02**	**1.77**	**9.23**	**1.38**	**8.85**
Copyin	**0.45**	**1.17**	**1.68**	**10.96**	**6.86**	**11.19**
ipintr	**0.19**	**0.93**	**0.94**	**6.31**	**4.88**	**5.11**
tcp_usrreq	2.20	2.20	1.93	2.00	2.03	2.24
tcp_input	**0.06**	**1.32**	**1.67**	**11.14**	**10.68**	**10.97**
tcp_output	**6.26**	**6.16**	**6.34**	**8.50**	**7.14**	**11.23**
tcp_xoutput	**0.00**	**0.38**	**0.28**	**2.11**	**1.68**	**5.45**
sbappend	**1.65**	**1.63**	**2.11**	**8.33**	**1.11**	**8.02**
ip_output	2.78	3.07	3.14	3.36	3.35	5.63
tcp_cksum	**2.23**	**3.13**	**3.65**	**16.16**	**10.52**	**16.72**
in_cksum	**1.89**	**1.72**	**1.57**	**3.69**	**2.66**	**4.41**
m_copy	**2.77**	**3.86**	**5.22**	**18.24**	**2.38**	**29.73**
enoutput	2.74	3.38	3.45	3.09	4.15	5.08
enstart	2.87	2.53	2.81	2.17	2.78	4.53
if_wubaput	**3.83**	**4.00**	**5.30**	**14.23**	**4.70**	**16.58**
in_lnaof	2.44	2.21	2.23	2.76	2.72	3.43
total of boldface routines	**24.21**	**30.08**	**36.77**	**124.43**	**59.52**	**144.78**
total of lightface routines	19.15	19.49	19.80	19.77	21.49	28.00

Table 1: Partial Decomposition of TCP/IP Processing Time in Seconds for 10,000 Transmissions Between two Dedicated VAX 11/750. [*Highlighted in boldface are those calls with larger timing changes.*]

gprof of TCP/IP	Message Size in Bytes					
	1	112	113	1023	1024	1025
Number of routines	233	266	260	264	265	265

Table 1a: Number of Different Routines in the Kernel **gprof** Profiling for TCP/IP.

As mentioned in Section 1, there are network hardware interfaces which provide checksumming facilities. As can be observed from the entries for *in_cksum* and *tcp_cksum*, the time spent in it is substantial. This, in fact, is true for both protocols (see Tables 1 and 2). It is clear, then, that redundant checksumming in a system has definite performance penalties.

4.2. UDP/IP

Table 2 presents the time spent in a selected group of routines that are called to process a UDP/IP datagram with a specific amount of data. These values were obtained directly from the *gprof* output.

From the user process viewpoint, sending data through UDP/IP results from system calls equivalent to *write* or *sendto*. *sendto* requires the destination address on each call; *sockargs* and *getsock* produce the socket control block associated with this operation. With this information, *sendit* is called, which in turn calls *sosend* as TCP/IP does. Again *sosend* calls *uiomove*, which

UDP/IP Routines and System Calls	Message Size in Bytes 1	112	113	1023	1024	1025
syscall	3.48	3.02	3.40	3.29	2.76	3.67
sendto	0.84	0.94	0.91	0.83	0.99	0.85
sockargs	0.80	0.93	0.87	0.81	0.77	0.84
getsock	0.79	0.62	0.46	0.63	0.64	0.71
sendit	2.92	2.84	2.94	2.85	2.69	2.46
m_freem	**2.59**	**2.35**	**2.98**	**3.20**	**4.43**	**2.28**
useracc	0.56	0.55	0.55	0.60	0.63	1.03
sosend	**5.07**	**5.75**	**6.68**	**19.25**	**5.58**	**7.38**
uiomove	**1.20**	**1.48**	**2.05**	**10.66**	**1.34**	**2.52**
Copyin	**1.14**	**2.02**	**2.37**	**14.58**	**8.45**	**8.47**
udp_usrreq	1.83	1.90	1.52	2.00	1.56	1.91
in_pcbconnect	2.24	2.45	2.13	2.56	2.65	2.17
in_pcblookup	0.95	1.13	0.81	1.28	0.96	1.03
in_netof	2.67	2.42	3.16	3.27	3.13	3.33
if_ifonnetof	0.46	0.46	0.53	0.64	0.73	0.62
in_pcbdisconnect	0.52	0.64	0.53	0.55	0.55	0.46
udp_output	2.61	2.45	2.28	3.66	3.07	2.78
m_get	**1.96**	**1.46**	**1.86**	**1.93**	**3.54**	**2.06**
ip_output	4.12	3.89	3.53	4.26	4.01	4.14
udp_cksum	**2.23**	**3.37**	**2.40**	**12.82**	**8.79**	**8.28**
in_cksum	4.19	4.54	4.43	3.44	3.56	4.01
in_lnaof	2.25	2.41	2.43	2.44	2.40	2.27
ipintr	4.42	3.71	3.96	2.69	4.34	4.36
enrint	3.07	3.51	4.44	3.37	3.90	4.14
enoutput	3.36	3.12	3.04	3.46	3.87	2.80
enstart	3.16	2.88	2.50	2.66	2.02	2.67
if_wubaput	**4.02**	**4.31**	**5.08**	**14.96**	**4.01**	**10.54**
getf	0.39	0.36	0.41	0.27	0.48	0.44
total of boldface routines	**18.21**	**20.74**	**23.42**	**77.40**	**36.14**	**41.53**
total of lightface routines	45.63	44.77	44.83	45.56	45.71	46.69

Table 2: Partial Decomposition of UDP/IP Processing Time in Seconds for Sending 10,000 Datagrams Between two Dedicated VAX 11/750. [*Highlighted in boldface are those calls with larger timing changes.*]

calls *Copyin* to actually copy the user data into buffers. These buffers are given to *udp_usrreq* which calls *udp_output* after a pseudo connection is established via *in_pcbconnect* with its associated routines, *in_pcblookup, in_netof, and if_ifonnetof*. As with TCP, *udp_output* represents the output processing of the UDP protocol. At this level the header is created and both the header and data are checksummed by *udp_cksum*. The header and the data are then passed to *ip_output* as in TCP/IP. *ip_output* checksums the header in *in_cksum* and passes the message to the appropriate network interface, in this case *en_output*. Again, the mbufs must be copied and mapped into a single contiguous memory space before transmission; this is done in *if_wubaput*.

From the row entries in Table 2, we can see that of the 28 different routines listed for UDP/IP, only 7 present processing costs which vary significantly with the amount of data sent. The processing time of the other 21 routines remains practically constant. In contrast, *m_freem* exhibits its largest processing time for datagrams of size 1024 bytes. The three calls which show a

gprof of UDP/IP	Message Size in Bytes					
	1	112	113	1023	1024	1025
Number of routines	261	252	261	254	256	255

Table 2a: Number of Different Routines in the Kernel **gprof** Profiling for UDP/IP.

larger variation in the vicinity of the 1024 bytes region are *uiomove*, as before with TCP/IP, *sosend*, and *if_wubaput*. The first two are associated with the buffer management strategy, and the latter with passing the data to be transmitted to the hardware interface in one contiguous piece. The four calls which have a larger impact in the processing of datagrams in UDP/IP are *udp_cksum*, *sosend*, *if_wubaput*, and *Copyin*. For UDP/IP, checksumming is, across most datagrams sizes, the single most expensive operation performed. As mentioned in Section 4.1, redundancy of this operation should be avoided if the Ethernet is reliable [20-21]. (However, TCP cannot avoid it.) For larger datagram sizes, the greatest impact comes from those routines which do copying of data across the interfaces. *sosend*, as was also the case in TCP/IP, is an important factor in the time spent processing messages.

Table 2 also shows that for UDP/IP's implementation the processing costs are somewhat more evenly distributed across many routines than for TCP/IP. This suggests that if checksumming is eliminated, speeding up more UDP/IP will require streamlining many routines. In UDP/IP there appear not to be many significant gains to be obtained from any one optimization.

TCP/IP Routines	Message Size in Bytes					
and System Calls	1	112	113	1023	1024	1025
syscall	10,882	10,884	10,882	10,884	10,882	10,882
write	*10,002*	*10,002*	*10,002*	*10,002*	*10,002*	*10,002*
rwuio	10,009	10,010	10,009	10,010	10,009	10,009
soo_rw	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
sosend	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
uiomove	**10,014**	**10,015**	**20,014**	**100,015**	**10,014**	**97,158**
ipintr	**356**	**1,532**	**1,686**	**10,035**	**10,238**	**10,012**
tcp_usrreq	10,011	10,020	10,021	10,119	10,022	10,046
tcp_input	**323**	**1,472**	**1,645**	**10,095**	**10,243**	**10,012**
tcp_output	10,010	10,017	10,015	10,104	10,008	10,015
tcp_xoutput	**323**	**1,472**	**1,645**	**10,095**	**10,243**	**10,012**
sbappend	10,000	10,000	10,000	10,088	10,000	10,000
ip_output	**10,016**	**10,049**	**10,032**	**10,182**	**10,024**	**20,081**
tcp_cksum	**10,333**	**11,497**	**11,671**	**20,234**	**20,256**	**30,037**
in_cksum	**10,384**	**11,628**	**11,730**	**20,494**	**20,356**	**30,369**
m_copy	**10,007**	**10,022**	**10,024**	**10,136**	**10,010**	**20,022**
enoutput	**10,016**	**10,049**	**10,032**	**10,182**	**10,024**	**20,081**
enstart	**10,024**	**10,059**	**10,037**	**10,188**	**10,057**	**20,141**
if_wubaput	**10,016**	**10,049**	**10,032**	**10,182**	**10,024**	**20,056**
in_lnaof	**30,075**	**30,190**	**30,131**	**30,644**	**30,136**	**60,361**

Table 3: Number of Calls per Function for 10,000 TCP/IP Transmissions.
[*Highlighted in boldface are those calls with large variations.*]
[*Highlighted in italics are those calls with identical counts.*]

4.3. Dynamic Profile of both Protocols

We can view each protocol's implementation in a different way by looking at the number of times each routine was called. Table 3 presents, for TCP/IP, such a decomposition, while Table 4 has it for UDP/IP. We first note that *sosend* is called by each protocol exactly 10,000 times. Moreover, *uiomove*, *ip_output*, *enoutput*, and *if_wubaput* are called about the same number of times for user data amounts of 1 through 1024 bytes. Indeed no buffering occurs for TCP since *ip_output* is called about the same number of times.

The 1025 byte size behaved quite differently. The most surprising behavior was *uiomove* in TCP/IP, where the count, instead of dropping to something in the order of 20,000, remained very high. This is so because of the 'stream' communication nature of TCP. As there are no record boundaries, the use of 'odd' send sizes causes *sosend* to copy user data in a number of small segments once the send queue reached the last 1024 bytes of buffer space allocated. If there is more than 1024 bytes of data to be transmitted but less than 1024 bytes of buffer space remaining for the socket, *sosend* will send the remaining amount using mbufs. This is exactly what happened in the 1025 byte case as witnessed by *uiomove* which is called once for each mbuf used. It is interesting to note that the UDP/IP implementation is more immune than the TCP/IP implementation to packet size changes, with respect to the number of times individual routines are called. This is mostly due to the fact that UDP preserves record boundaries and has no flow control provisions. In TCP, estimating the amount of data to copy into mbufs based solely on the amount of buffer space currently available is the problem. In the UDP/IP implementation, however, *m_freem* and *m_get* exhibit a peak of activity for the 1024 byte case which contrasts with all other sizes. This is due to the handling of trailer protocol packets.

5. Conclusions

For users who want to implement distributed applications based on Berkeley UNIX 4.2BSD computing environments interconnected through Ethernets, the system currently provides two basic transmission protocols for interprocess communication: TCP and UDP. Both, in turn, are based on IP for actual data transmissions. This paper has presented a microanalysis of the dynamic behavior that the current implementation of these protocols exhibits. Even though there are currently other protocol families at different stages of implementation will coexist with the above protocols in the kernel of Berkeley UNIX, this paper has only addressed performance issues relating to TCP/IP and UDP/IP.

Let us define user process network latency to be the minimum time required to send a single byte of data. When the ether and both the sending and receiving hosts had no other user activities in them but our tests, complementary results [2] show that, for the VAX 11/750, the latency for TCP/IP is approximately 5.5 milliseconds, and for UDP/IP is 6.4 milliseconds. Sending 1024 bytes took 13.3 milliseconds with TCP/IP and 7.8 milliseconds with UDP/IP. We thus see that the transmission cost per byte is substantially lower for 1024 byte messages. Moreover, because of packet acknowledgements TCP/IP is sensitive to round trip network time while UDP/IP is not.

A detailed protocol implementation analysis has been presented for TCP/IP and UDP/IP. For TCP/IP, those routines which do the copying of data appear to make preponderant contributions to the total elapsed time (see Table 1). For UDP/IP, the single most expensive operation is the computation of the checksums (see Table 2). For both protocols, the buffer scheme used in the implementation appears to have an overwhelming effect on performance. Since UDP/IP has no flow control, it sends data atomically and only limits a packet's maximum size. This protocol is not so sensitive to varying data sizes. The drastic increases in overhead in the routines shown in bold appear to be due to the data buffer management scheme chosen. On the other hand, TCP/IP, with its windows and data streaming, is sensitive to the amount of data presented. Thus, in addition to the increased overhead seen in the routines which deal with data within the buffer management system, the actual protocol implementation overhead appears to increase noticeably because of the varying amounts of data presented.

UDP/IP Routines and System Calls	Message Size in Bytes 1	112	113	1023	1024	1025
syscall	10,886	10,888	10,886	10,888	10,886	10,886
sendto	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
sockargs	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
getsock	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
sendit	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
m_freem	**19,003**	**19,406**	**19,204**	**19,331**	**29,194**	**19,848**
sosend	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
uiomove	**10,015**	**10,016**	**20,015**	**100,016**	**10,015**	**20,015**
udp_usrreq	*10,002*	*10,002*	*10,002*	*10,002*	*10,002*	*10,002*
in_pcbconnect	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
in_pcblookup	10,052	10,055	10,049	10,089	10,069	10,073
in_netof	40,057	40,051	40,075	40,108	40,075	40,111
if_ifonnetof	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
in_pcbdisconnect	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
udp_output	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
m_get	**20,023**	**20,021**	**20,030**	**20,040**	**30,029**	**20,041**
ip_output	10,019	10,017	10,025	10,036	10,025	10,057
udp_cksum	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
in_cksum	29,021	28,820	28,458	28,710	29,818	29,770
in_lnaof	30,108	30,105	30,122	30,196	30,143	30,183
ipintr	8,996	9,403	9,211	9,347	9,888	9,850
enrint	8,255	8,503	8,298	8,369	8,117	8,136
enoutput	10,019	10,017	10,025	10,036	10,025	10,037
enstart	10,037	10,039	10,037	10,056	10,028	10,044
if_wubaput	10,019	10,017	10,025	10,036	10,025	10,037
getf	*10,840*	*10,840*	*10,840*	*10,840*	*10,840*	*10,840*

Table 4: Number of Calls per Function for 10,000 UDP/IP Datagrams.
[*Highlighted in boldface are those calls with large variations.*]
[*Highlighted in italics are those calls with identical counts.*]

The miscoordination of buffering between different layers with independent data placement policies can lead to severe inefficiencies. This is best exemplified by the behavior of *sosend* when transmitting 1025 byte messages under TCP/IP. The strategy of completely filling the send buffer, despite the induced buffer fragmentation, rather than waiting for additional buffer availability is inadequate.

6. Epilogue

Since this study was conducted several changes have been made to the implementations of TCP/IP and UDP/IP, as well as to the buffer management policies and default buffer sizes. These changes will be present in future BSD releases. We highlight some.

The buffer size at the socket level is now a settable parameter. When increased to 8 kilobytes we observed an improved throughput for TCP/IP in the order of 20%. In TCP, a facility has been added for coalescing messages to be sent while waiting for acknowledgement of outstanding packets. *sosend* has been changed so as to align 1024 byte messages whenever possible, delaying if necessary. From the receiver end of transmissions, and having in mind that the processing of acknowledgements consumes a substantial amount of processor resources, the scheme for delayed acknowledgements has been tuned. This scheme works best with a larger socket buffer size. For UDP, routing has been enhanced to cache the last computed route; if two

consecutive datagrams go to the same destination, the route for the second need not be computed. This routing change should improve throughput for multiple datagrams to the same destination. A complete assessment of these changes has yet to be made.

7. Bibliography

[1] Almes, G. T., and Lazowska, E. D., "The Behavior of Ethernet-like Computer Communications Networks", Proceedings of the 7th Symposium on Operating System Principles, 1979, pp. 66-81.

[2] Cabrera, L. F., Hunter, E., Karels, M. J., and Mosher, D., "A User-Process Oriented Performance Study of Ethernet Networking Under Berkeley UNIX 4.2BSD," Report No. UCB/CSD 84/217, University of California, Berkeley, December 1984, pp. 1-37.

[3] Cheriton, D., and Zwaenepoel, W., "The Distributed V Kernel and its Performance for Diskless Workstations", Proceedings of the 9th SOSP, November 1983.

[4] Gonsalves, T. M., "Packet-Voice Communication on an Ethernet Local Computer Network: an Experimental Study", Proceedings of the SIGCOMM 1983 Symposium on Communication Architectures and Protocols, Austin, Texas, March 1983, pp. 178-185.

[5] Graham, S. L., Kessler, P. B., and McKusick, M. K., "An Execution Profiler for Modular Programs", Software -- Practice and Experience, Vol. 13, 1983, pp. 671-685.

[6] Hagmann, R. B., "Performance Analysis of Several Backend Database Architectures", Ph.D. Thesis, Report No. UCB/CSD 83/124, University of California, Berkeley, August 1983.

[7] Hunter, E., "A Performance Study of the Ethernet Under Berkeley UNIX 4.2BSD", Proceedings of CMG XV, San Francisco, California, December 1984, pp. 373-382.

[8] Lazowska, E. D., Zahorjan, J., Zwaenepoel, W., and Cheriton, D. R., "File Access Performance of Diskless Workstations", Technical Report 84-06-01, June 1984, Department of Computer Science, University of Washington.

[9] Leffler, S. J., and Fabry, R., "A 4.2BSD Interprocess Communication Primer", Report No. UCB/CSD 83/145, University of California, Berkeley, July 1983.

[10] Leffler, S. J., Joy, W. N., and Fabry, R. S., "4.2BSD Networking Implementation Notes," Report No. UCB/CSD 83/146, University of California, Berkeley, July 1983.

[11] Leffler, S. J., Karels, M., and McKusick, M. K., "Measuring and Improving the Performance of 4.2BSD", Proceedings of the Summer 1984 USENIX Conference, Salt Lake City, June 1984, pp. 237-252.

[12] Leffler, S. J., and Karels, M. J., "Trailer Encapsulations", RFC 893, Network Working Group, University of California, Berkeley, April 1984.

[13] Metcalfe, R. M., and Boggs, D. R., "Ethernet: Distributed Packet Switching for Local Computer Networks", CACM, Volume 19, Number 7, July 1976, pp. 395-404.

[14] Padlipsky, M., "TCP-ON-A-LAN", RFC 872, USC Information Sciences Institute, September 1982.

[15] Postel, J., "User Datagram Protocol", RFC 768, USC Information Sciences Institute, August 1980.

[16] Postel, J., "Internet Protocol - DARPA Internet Program Protocol Specification", RFC 791, USC Information Sciences Institute, September 1981.

[17] Postel, J., "Transmission Control Protocol", RFC 793, USC Information Sciences Institute, September 1981.

[18] Rajaraman, M. K., "Performance Measures for a Local Network", ACM Sigmetrics Performance Evaluation Review, Volume 12, Number 2, Spring-Summer 1984, pp. 34-37.

[19] Sechrest, S., "Tutorial Examples of Interprocess Communication in Berkeley UNIX 4.2BSD", Report No. UCB/CSD 84/191, University of California, Berkeley, June 1984.

[20] Shoch, J. F., and Hupp, J. A., "Performance of an Ethernet Local Network -- A Preliminary Report", Proceedings of Spring COMPCON 80, San Francisco, February 1980.

[21] Shoch, J. F., and Hupp, J. A., "Measured Performance of an Ethernet Local Network", CACM, Volume 23, Number 12, December 1980, pp. 711-721.

[22] Terry, D., and Andler, S., "Experience With Measuring Performance of Local Network Communications", IBM San Jose Research Laboratory Research Report RJ 3743 (43119), December 1983, pp. 1-6.

[23] "UNIX Programmer's Manual", 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, Computer Science Division, University of California at Berkeley, August 1983.

8. Appendix A

8.1. Software for TCP/IP Assessment (Sender)

```
#include <sys/types.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>

struct sockaddr_in address;
char sendbuf[1025];;

main(argc,argv)
char *argv[];
{
        int i;
        struct hostent *phostent;
        int des;
        int size;

        des = socket(AF_INET,SOCK_STREAM,0);
        if (des < 0 )
                perror("socket"),exit(-1);
        phostent = gethostbyname(argv[1]);
        address.sin_family = AF_INET;
        address.sin_addr.s_addr = *(int *)phostent->h_addr;
        address.sin_port = 4321;
        if (connect(des,&address,sizeof(address)))
                perror("connect"), exit(-1);
        size = atoi(argv[2]);
        for (i=0; i<10000; i++)
                write(des,sendbuf,size);
}
```

8.2. Software for UDP/IP Assessment

```
#include <sys/types.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>

struct sockaddr_in address;

main(argc,argv)
char *argv[];
{
        int s, i;
        char buf[1025];
        struct hostent *phostent;
        int size;
        extern errno;

        if((s = socket(AF_INET,SOCK_DGRAM,0)) == -1) perror("socket") ;
        phostent = gethostbyname(argv[1]);
        address.sin_family = AF_INET;
        address.sin_addr.s_addr = *(int *)phostent->h_addr;
        address.sin_port = ntohs(1234);
        size = atoi(argv[2]);
        printf("Testing byte size of %d0,size);
        for(i = 0; i < 10000; i++)
                sendto(s,buf,size,0,&address,sizeof(address));
        if (errno) perror("udp: ");
}
```

Performance Improvements and Functional Enhancements in 4.3BSD

M. Kirk McKusick
Mike Karels

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

Sam Leffler

Computer Division
Lucasfilm, Ltd.
PO Box 2009
San Rafael, California 94912

ABSTRACT

The 4.2 Berkeley Software Distribution of UNIX† for the VAX‡ provided many new facilities. This paper highlights the changes to 4.2BSD that appear in 4.3BSD. The changes to the system have consisted of improvements to the performance of the existing facilities, as well as enhancements to the current facilities. Performance improvements in the kernel include cacheing of path name translations, reductions in clock handling and scheduling overhead, and improved throughput of the network. Performance improvements in the libraries and utilities include replacement of linear searches of system databases with indexed lookup, merging of most network services into a single daemon, and conversion of system utilities to use the more efficient facilities available in 4.2BSD. Enhancements in the kernel include the addition of subnets and gateways, increases in many kernel limits, cleanup of the signal and autoconfiguration implementations, and support for windows and system logging. Functional extensions in the libraries and utilities include the addition of an Internet name server, new system management tools, and extensions to *dbx* to work with Pascal. The paper concludes with a brief discussion of changes made to the system to enhance security.

† UNIX is a Trademark of Bell Laboratories.

‡ VAX, MASSBUS, UNIBUS, and DEC are trademarks of Digital Equipment Corporation.

1. Introduction

The development effort for 4.2BSD concentrated on providing new facilities, and in getting them to work correctly. The limited development period left little time for tuning the completed system. The increased user feedback that came with the release of 4.2BSD and a growing body of experience with the system highlighted the performance shortcomings of 4.2BSD. With the release of 4.3BSD many of these problems have been addressed. The first part of this paper describes the performance improvements that have been made to the system; the second part describes functional enhancements that have been made; the final part discusses some of the security problems that have been addressed. Much of the work has been done in the machine independent parts of the system, hence these improvements could be applied to other variants of UNIX with equal success.

2. Performance Improvements

Techniques for measuring, benchmarking, and tuning 4.2BSD are described in [Leffler84]. The purpose of this paper is to report on more recent results of those efforts and to describe the implementations that we incorporated in the 4.3BSD release. The improvements fall into two major classes; changes to the kernel that are described in this section, and changes to the system libraries and utilities that are described in the following section.

2.1 Performance Improvements in the Kernel

Our goal has been to optimize system performance for our general timesharing environment. Since most sites running 4.2BSD have been forced to take advantage of declining memory costs rather than replace their existing machines with ones that are more powerful, we have chosen to optimize running time at the expense of memory. This tradeoff may need to be reconsidered for personal workstations that have smaller memories and higher latency disks. Decreases in the running time of the system may be unnoticeable because of higher paging rates incurred by a larger kernel. Where possible, we have allowed the size of caches to be controlled so that systems with limited memory may reduce them as appropriate.

Problem areas are identified by running long term profiling studies on general time sharing machines. Once a particular problem area is identified a "micro benchmark" is written to exercise a particular feature. When the kernel has been optimized for the benchmark, a long term profiling study is run to verify that the change works in normal use. Figure 1 describes several micro benchmarks that measure the speed of common operations handled by the kernel. Figure 2 shows the running time of the micro benchmarks on a 2 megabyte VAX 11/750 running 4.1BSD, 4.2BSD, and 4.3BSD.

The major changes to the kernel are described in the following subsections.

2.1.1 Name Cacheing

Our initial profiling studies showed that more than one quarter of the time in the system was spent translating path names to inodes[1]. More detailed analysis showed two major sources of names to be translated. One major source of name translations came from requests to execute system utilities or access system databases such as the password file. The other major source of names to be translated came from programs making a linear scan of a directory.

[1] Inode is an abbreviation for "Index node". Each file on the system is described by an inode; the inode maintains access permissions, and an array of pointers to the disk blocks that hold the data associated with the file.

Test	Description
syscall	perform 100,000 *getpid* system calls
csw	perform 10,000 context switches using signals
signocsw	send 10,000 signals to yourself
pipeself4	send 10,000 4-byte messages to yourself
pipeself512	send 10,000 512-byte messages to yourself
pipediscard4	send 10,000 4-byte messages to child who discards
pipediscard512	send 10,000 512-byte messages to child who discards
pipeback4	exchange 10,000 4-byte messages with child
pipeback512	exchange 10,000 512-byte messages with child
forks0	fork-exit-wait 1,000 times
forks1k	sbrk(1024), fault page, fork-exit-wait 1,000 times
forks100k	sbrk(102400), fault pages, fork-exit-wait 1,000 times
vforks0	vfork-exit-wait 1,000 times
vforks1k	sbrk(1024), fault page, vfork-exit-wait 1,000 times
vforks100k	sbrk(102400), fault pages, vfork-exit-wait 1,000 times
execs0null	fork-exec "null job"-exit-wait 1,000 times
execs0null (1K env)	execs0null above, with 1K environment added
execs1knull	sbrk(1024), fault page, fork-exec "null job"-exit-wait 1,000 times
execs1knull (1K env)	execs1knull above, with 1K environment added
execs100knull	sbrk(102400), fault pages, fork-exec "null job"-exit-wait 1,000 times
vexecs0null	vfork-exec "null job"-exit-wait 1,000 times
vexecs1knull	sbrk(1024), fault page, vfork-exec "null job"-exit-wait 1,000 times
vexecs100knull	sbrk(102400), fault pages, vfork-exec "null job"-exit-wait 1,000 times
execs0big	fork-exec "big job"-exit-wait 1,000 times
execs1kbig	sbrk(1024), fault page, fork-exec "big job"-exit-wait 1,000 times
execs100kbig	sbrk(102400), fault pages, fork-exec "big job"-exit-wait 1,000 times
vexecs0big	vfork-exec "big job"-exit-wait 1,000 times
vexecs1kbig	sbrk(1024), fault pages, vfork-exec "big job"-exit-wait 1,000 times
vexecs100kbig	sbrk(102400), fault pages, vfork-exec "big job"-exit-wait 1,000 times

Figure 1. Benchmark programs.

Frequent requests for a small set of names are best handled with a cache of recent name translations. The system already maintained a cache of recently accessed inodes, so the initial name cache maintained a simple name-inode association that was used to check each component of a path name during name translations. We considered implementing the cache by tagging each inode with its most recently translated name, but eventually decided to have a separate data structure that kept names with pointers to the inode table. Tagging inodes has two drawbacks; many inodes such as those associated with login ports remain in the inode table for a long period of time, but are never looked up by name. Other inodes, such as those describing directories are looked up frequently by many different names (*e.g.* ".."). By keeping a separate table of names, the cache can truly reflect the most recently used names. An added benefit is that the table can be sized independently of the inode table, so that machines with small amounts of memory can reduce the size of the cache (or even eliminate it) without modifying the inode table structure.

Test	Real			User			System		
	4.1	4.2	4.3	4.1	4.2	4.3	4.1	4.2	4.3
syscall	28.0	29.0	23.0	4.5	5.3	3.5	23.9	23.7	20.4
csw	45.0	60.0	45.0	3.5	4.3	3.3	19.5	25.4	19.0
signocsw	16.5	23.0	16.0	1.9	3.0	1.1	14.6	20.1	15.2
pipeself4	21.5	29.0	26.0	1.1	1.1	0.8	20.1	28.0	25.6
pipeself512	47.5	59.0	55.0	1.2	1.2	1.0	46.1	58.3	54.2
pipediscard4	32.0	42.0	36.0	3.2	3.7	3.0	15.5	18.8	15.6
pipediscard512	61.0	76.0	69.0	3.1	2.1	2.0	29.7	36.4	33.2
pipeback4	57.0	75.0	66.0	2.9	3.2	3.3	25.1	34.2	29.7
pipeback512	110.0	138.0	125.0	3.1	3.4	2.2	52.2	65.7	57.7
forks0	37.5	41.0	22.0	0.5	0.3	0.3	34.5	37.6	21.5
forks1k	40.0	43.0	22.0	0.4	0.3	0.3	36.0	38.8	21.6
forks100k	217.5	223.0	176.0	0.7	0.6	0.4	214.3	218.4	175.2
vforks0	34.5	37.0	22.0	0.5	0.6	0.5	27.3	28.5	17.9
vforks1k	35.0	37.0	22.0	0.6	0.8	0.5	27.2	28.6	17.9
vforks100k	35.0	37.0	22.0	0.6	0.8	0.6	27.6	28.9	17.9
execs0null	97.5	92.0	66.0	3.8	2.4	0.6	68.7	82.5	48.6
execs0null (1K env)	197.0	229.0	75.0	4.1	2.6	0.9	167.8	212.3	62.6
execs1knull	99.0	100.0	66.0	4.1	1.9	0.6	70.5	86.8	48.7
execs1knull (1K env)	199.0	230.0	75.0	4.2	2.6	0.7	170.4	214.9	62.7
execs100knull	283.5	278.0	216.0	4.8	2.8	1.1	251.9	269.3	202.0
vexecs0null	100.0	92.0	66.0	5.1	2.7	1.1	63.7	76.8	45.1
vexecs1knull	100.0	91.0	66.0	5.2	2.8	1.1	63.2	77.1	45.1
vexecs100knull	100.0	92.0	66.0	5.1	3.0	1.1	64.0	77.7	45.6
execs0big	129.0	201.0	101.0	4.0	3.0	1.0	102.6	153.5	92.7
execs1kbig	130.0	202.0	101.0	3.7	3.0	1.0	104.7	155.5	93.0
execs100kbig	318.0	385.0	263.0	4.8	3.1	1.1	286.6	339.1	247.9
vexecs0big	128.0	200.0	101.0	4.6	3.5	1.6	98.5	149.6	90.4
vexecs1kbig	125.0	200.0	101.0	4.7	3.5	1.3	98.9	149.3	88.6
vexecs100kbig	126.0	200.0	101.0	4.2	3.4	1.3	99.5	151.0	89.0

Table 2. Benchmark results (all times in seconds).

Another issue to be considered is how the name cache should hold references to the inode table. Normally processes hold "hard references" by incrementing the reference count in the inode they reference. Since the system reuses only inodes with zero reference counts, a hard reference insures that the inode pointer will remain valid. However, if the name cache holds hard references, it is limited to some fraction of the size of the inode table, since some inodes must be left free for new files. It also makes it impossible for other parts of the kernel to verify sole use of a device or file. These reasons made it impractical to use hard references without affecting the behavior of the inode cacheing scheme. Thus, we chose instead to keep "soft references" protected by a *capability* — a 32-bit number guaranteed to be unique[2]. When an entry is made in the name cache, the capability of its inode is copied to the name cache entry. When an inode is reused it is issued a new capability. When a name cache hit occurs, the capability of the name cache entry is compared with the capability of the inode that it references. If the capabilities do not match, the name cache entry is invalid. Since the name cache holds only soft references, it may be sized independent of the size of the inode table. A final benefit of using capabilities is that all cached names for an inode may be invalidated without searching through the entire cache; instead all you need to do is assign a new capability to the inode.

[2] When all the numbers have been exhausted, all outstanding capabilities are purged and numbering starts over from scratch. Purging is possible as all capabilities are easily found in kernel memory.

The name cache is able to resolve more than 70% of the names passed to the system. However, it does not provide any help for programs that sequentially scan a directory. These names are not usually in the cache and, once operated on, they are unlikely to be accessed again. To improve performance for processes doing directory scans, the system keeps track of the directory offset of the last component of the most recently translated path name for each process. If the next name the process requests is in the same directory, the search is started from the point that the previous name was found (instead of from the beginning of the directory). This changes the previous $O(n^2)$ algorithm to a (potentially) $O(n)$ algorithm.

On our general time sharing systems we find that during the twelve hour period from 8AM to 8PM the system does 500,000 to 1,000,000 name translations. The name cache has a hit rate of 70%-80%; the directory offset cache gets a hit rate of 5%-15%. The combined hit rate of the two caches almost always adds up to 85%. With the addition of the two caches, the percentage of system time devoted to name translation has dropped from 25% to less than 10%.

2.1.2 Intelligent Auto Siloing

Most VAX terminal input hardware can run in two modes: it can either generate an interrupt each time a character is received, or collect characters in a silo that the system then periodically drains. To provide quick response for interactive input and flow control, a silo must be checked 30 to 50 times per second. Ascii terminals normally exhibit an input rate of less than 30 characters per second so, they are most efficiently handled with interrupt per character mode. When input is being generated by another machine, however, the input rate is usually more than 50 characters per second so, it is more efficient to use a device's silo input mode. Since a given dialup port may switch between uucp logins and user logins, it is impossible to statically select the most efficient input mode to use. Thus, the system monitors the character input rate; input is handled on an interrupt at a time basis during periods of low input rates, and with the hardware silos during periods of high input rates.

2.1.3 Process Table Management

As systems have grown larger, the size of the process table has grown far past 200 entries. With large tables, linear searches must be eliminated from any frequently used facility. The kernel process table is now multi-threaded to allow selective searching of active and zombie processes. A third list threads unused process table slots. Free slots can be obtained in constant time by taking one from the front of the free list. The number of processes used by a given user may be computed by scanning only the active list. Since the 4.2BSD release, the kernel maintained linked lists of the descendents of each process. This linkage is now exploited when dealing with process exit; parents seeking the exit status of children now avoid linear search of the process table, but examine only their direct descendents. In addition, the previous algorithm for finding all descendents of an exiting process used multiple linear scans of the process table. This has been changed to follow the links between child process and siblings.

When forking a new process, the system must assign it a unique process identifier. The system previously scanned the entire process table each time it created a new process to locate an identifier that was not already in use. Now, to avoid scanning the process table for each new process, the system computes a range of unused identifiers that can be directly assigned. Only when the set of identifiers is exhausted is another process table scan required.

2.1.4 Scheduling

Previously the scheduler scanned the entire process table once per second to recompute process priorities. Processes that had run for their entire time slice had their priority lowered. Processes that had not used their time slice, or that had been sleeping for the past second had their priority raised. On systems running many processes, the scheduler represented nearly 20% of the system time. To reduce this overhead, the scheduler has been changed to consider only runnable processes when recomputing priorities. To insure that processes sleeping for more than

a second still get their appropriate priority boost, their priority is recomputed when they are placed back on the run queue. Since the set of runnable process is typically only a small fraction of the total number of processes on the system, the cost of invoking the scheduler drops proportionally.

2.1.5 Clock Handling

The hardware clock interrupts the processor 100 times per second at high priority. As most of the clock-based events need not be done at high priority, the system schedules a lower priority software interrupt to do the less time-critical events such as cpu scheduling and timeout processing. Often there are no such events, and the software interrupt handler finds nothing to do and returns. The high priority event now checks to see if there are low priority events to process; if there is nothing to do, the software interrupt is not requested. Often, the high priority interrupt occurs during a period when the machine had been running at low priority. Rather than posting a software interrupt that would occur as soon as it returns, the hardware clock interrupt handler simply lowers the processor priority and calls the software clock routines directly. Between these two optimizations, nearly 80 of the 100 software interrupts per second can be eliminated.

2.1.6 File System

The file system uses a large block size, typically 4096 or 8192 bytes. To allow small files to be stored efficiently, the large blocks can be broken into smaller fragments, typically multiples of 1024 bytes. To minimize the number of full-sized blocks that must be broken into fragments, the file system uses a best fit strategy. Programs that slowly grow files using write of 1024 bytes or less can force the file system to copy the data to successively larger and larger fragments until it finally grows to a full sized block. The file system still uses a best fit strategy the first time a fragment is written. However, the first time that the file system is forced to copy a growing fragment it places it at the beginning of a full sized block. Continued growth can be accommodated without further copying by using up the rest of the block. If the file ceases to grow, the rest of the block is still available for holding other fragments.

When creating a new file name, the entire directory in which it will reside must be scanned to insure that the name does not already exist. For large directories, this scan is time consuming. Because there was no provision for shortening directories, a directory that is once over-filled will increase the cost of file creation even after the over-filling is corrected. Thus, for example, a congested uucp connection can leave a legacy long after it is cleared up. To alleviate the problem, the system now deletes empty blocks that it finds at the end of a directory while doing a complete scan to create a new name.

2.1.7 Network

The default amount of buffer space allocated for stream sockets (including pipes) has been increased to 4096 bytes. Stream sockets and pipes now return their buffer sizes in the block size field of the stat structure. This information allows the standard I/O library to use more optimal buffering. Unix domain stream sockets also return a dummy device and inode number in the stat structure to increase compatibility with other pipe implementations. The TCP maximum segment size is calculated according to the destination and interface in use; non-local connections use a more conservative size for long-haul networks.

On multiply-homed hosts, the local address bound by TCP now always corresponds to the interface that will be used in transmitting data packets for the connection. Several bugs in the calculation of round trip timing have corrected. TCP now switches to an alternate gateway when an existing route fails, or when an ICMP redirect message is received. ICMP source quench messages are used to throttle the transmission rate of TCP streams by temporarily creating an artificially small send window, and retransmissions send only a single packet rather than resending all queued data. A send policy has been implemented that decreases the number of small packets outstanding for network terminal traffic [Nagle84], providing additional reduction of network

congestion. The overhead of packet routing has been decreased by changes in the routing code and by cacheing the most recently used route for each datagram socket.

2.1.8 Exec

When *exec*-ing a new process, the kernel creates the new program's argument list by copying the arguments and environment from the parent process's address space into the system, then back out again onto the stack of the newly created process. These two copy operations were done one byte at a time, but are now done a string at a time. This optimization reduced the time to process an argument list by a factor of ten; the average time to do an *exec* call decreased by 25%.

2.1.9 Context Switching

The kernel used to post a software event when it wanted to force a process to be rescheduled. Often the process would be rescheduled for other reasons before exiting the kernel, delaying the event trap. At some later time the process would again be selected to run and would complete its pending system call, finally causing the event to take place. The event would cause the scheduler to be invoked a second time selecting the same process to run. The fix to this problem is to cancel any software reschedule events when saving a process context. This change doubles the speed with which processes can synchronize using pipes or signals.

2.1.10 Setjmp/Longjmp

The kernel routine *setjmp*, that saves the current system context in preparation for a nonlocal goto used to save many more registers than necessary under most circumstances. By trimming its operation to save only the minimum state required, the overhead for system calls decreased by an average of 13%.

2.1.11 Compensating for Lack of Compiler Technology

The current compilers available for C do not do any significant optimization. Good optimizing compilers are unlikely to be built; the C language is not well suited to optimization because of its rampant use of unbound pointers. Thus, many classical optimizations such as common subexpression analysis and selection of register variables must be done by hand using "exterior" knowledge of when such optimizations are safe.

Another optimization usually done by optimizing compilers is inline expansion of small or frequently used routines. In past Berkeley systems this has been done by using *sed* to run over the assembly language and replace calls to small routines with the code for the body of the routine, often a single VAX instruction. While this optimization eliminated the cost of the subroutine call and return, it did not eliminate the pushing and popping of several arguments to the routine. The *sed* script has been replaced by a more intelligent expander, *inline*, that merges the pushes and pops into moves to registers. For example, if the C code

```
if (scanc(map[i], 1, 47, i - 63))
```

is compiled into assembly language it generates the code shown in the left hand column of Figure 3. The *sed* inline expander changes this code to that shown in the middle column. The newer optimizer eliminates most of the stack operations to generate the code shown in the right hand column.

cc		sed		inline	
subl3	$64,__i,—(sp)	subl3	$64,__i,—(sp)	subl3	$64,__i,r5
pushl	$47	pushl	$47	movl	$47,r4
pushl	$1	pushl	$1	pushl	$1
mull2	$16,__i,r3	mull2	$16,__i,r3	mull2	$16,__i,r3
pushl	—56(fp)[r3]	pushl	—56(fp)[r3]	movl	—56(fp)[r3],r2
calls	$4,__scanc	movl	(sp)+,r5	movl	(sp)+,r3
tstl	r0	movl	(sp)+,r4	scanc	r2,(r3),(r4),r5
jeql	L7	movl	(sp)+,r3	tstl	r0
		movl	(sp)+,r2	jeql	L7
		scanc	r2,(r3),(r4),r5		
		tstl	r0		
		jeql	L7		

Figure 3. Inline code expansion.

Another optimization involved reevaluating existing data structures in the context of the current system. For example, disk buffer hashing was implemented when the system typically had thirty to fifty buffers. Most systems today have 200 to 1000 buffers. Consequently, most of the hash chains contained ten to a hundred buffers each! The running time of the low level buffer management primitives was dramatically improved simply by enlarging the size of the hash table.

2.2 Improvements to Libraries and Utilities

Intuitively, changes to the kernel would seem to have the greatest payoff since they affect all programs that run on the system. However, the kernel has been tuned many times before, so the opportunity for significant improvement is small. By contrast, many of the libraries and utilities have never been tuned. For example, we have found utilities that spent 90% of their running time doing single character read system calls. Changing the utility to use the standard I/O library cut the running time by a factor of five! Thus, while most of our time has been spent tuning the kernel, more than half of the speedups are because of improvements in other parts of the system. Some of the more dramatic changes are described in the following subsections.

2.2.1 Hashed Databases

UNIX provides a set of database management routines, *dbm*, that can be used tc speed lookups in large data files with an external hashed index file. The original version of dbm was designed to work with only one database at a time. These routines were generalized to handle multiple database files, enabling them to be used in rewrites of the password and host file lookup routines. The new routines used to access the password file significantly improve the running time of many important programs such as the mail subsystem, the C-shell (in doing tilde expansion), *ls —l*, etc.

2.2.2 Buffered I/O

The standard error file (**stderr**) is now buffered to do only a single write for each call into the standard I/O library routines (*e.g.* fprintf). This buffering still allows partial line writes, but without requiring character at a time output that can congest a network. Several important utilities that did not use the standard I/O library and were buffering I/O using the old optimal buffer size, 1Kbytes; the programs were changed to buffer I/O according to the optimal file system blocksize. These include the editor, the assembler, loader, remote file copy, the text formatting programs, and the C compiler.

2.2.3 Mail System

The mail system previously used *link* and *unlink* in implementing file locking primitives. Because these operations usually modify the contents of directories they require synchronous disk operations and cannot take advantage of the name cache maintained by the system. Unlink requires that the entry be found in the directory so that it can be removed; link requires that the directory be scanned to insure that the name does not already exist. By contrast the advisory locking facility in 4.2BSD is efficient because it is all done with in-memory tables. Thus, the mail system was modified to use the file locking primitives; it also benefited from the database hashing and extensive profiling and tuning of *sendmail*.

2.2.4 Network Servers

With the introduction of the network facilities in 4.2BSD, a myriad of services became available, each of which required its own daemon process. Many of these daemons were rarely if ever used, yet they lay asleep in the process table consuming system resources and generally slowing down response. Most of the daemons were eliminated by merging them into a single "Internet daemon" that listens on all the service ports and only forks a server process when a request for their service arrives. This allowed as many as twenty processes to be eliminated.

2.2.5 The C Run-time Library

Several people have found poorly tuned code in frequently used routines in the C library [Lankford84]. In particular the running time of the string routines can be cut in half by rewriting them using the VAX string instructions. The memory allocation routines have been tuned to waste less memory for memory allocations with sizes that are a power of two. Certain library routines that did file input in one-character reads have been corrected. Other library routines including *fread* and *fwrite* have been rewritten for efficiency.

2.2.6 Csh

The C-shell was converted to run on 4.2BSD by writing a set of routines to simulate the old jobs library. While this provided a functioning C-shell, it was grossly inefficient, generating up to twenty system calls per prompt. The C-shell has been modified to use the new signal facilities directly, cutting the number of system calls per prompt in half. Additional tuning was done with the help of profiling to cut the cost of frequently used facilities.

3. Functional Extensions

Some of the facilities introduced in 4.2BSD were not completely implemented. An important part of the effort that went into 4.3BSD was to clean up and unify both new and old facilities.

3.1 Kernel Extensions

A significant effort went into improving the networking part of the kernel. The work consisted of fixing bugs, tuning the algorithms, and revamping the lowest levels of the system to better handle heterogeneous network topologies.

3.1.1 Subnets, Broadcasts and Gateways

To allow sites to expand their network in an autonomous and orderly fashion, subnetworks have been introduced in 4.3BSD [GADS85]. This facility allows sites to subdivide their local Internet address space into multiple subnetwork address spaces that are visible only by hosts at that site. To off-site hosts machines on a site's subnetworks appear to reside on a single network. The routing daemon has been reworked to provide routing support in this type of environment.

The default Internet broadcast address is now specified with a host part of all one's, rather than all zero's. The broadcast address may be set at boot time on a per-interface basis.

3.1.2 Interface Addressing

The organization of network interfaces has been reworked to more cleanly support multiple network protocols. Network interfaces no longer contain a host's address on that network; instead each interface contains a pointer to a list of addresses assigned to that interface. This permits a single interface to support, for example, Internet protocols at the same time as XNS protocols.

The Address Resolution Protocol (ARP) support for 10 megabyte/second Ethernet† has been made more flexible by allowing hosts to act as an "clearing house" for hosts that do not support ARP. In addition, system managers have more control over the contents of the ARP translation cache and may interactively interrogate and modify the cache's contents.

3.1.3 User Control of Network Buffering

Although the system allocates reasonable default amounts of buffering for most connections, certain operations such as file system dumps to remote machines benefit from significant increases in buffering [Walsh84]. The *setsockopt* system call has been extended to allow such requests. In addition, *getsockopt* and *setsockopt*, are now interfaced to the protocol level allowing protocol-specific options to be manipulated by the user.

3.1.4 Number of File Descriptors

To allow full use of the many descriptor based services available, the previous hard limit of 30 open files per process has been relaxed. The changes entailed generalizing *select* to handle arrays of 32-bit words, removing the dependency on file descriptors from the page table entries, and limiting most of the linear scans of a process's file table. The default per-process descriptor limit was raised from 20 to 64, though there are no longer any hard upper limits on the number of file descriptors.

3.1.5 Kernel Limits

Many internal kernel configuration limits have been increased by suitable modifications to data structures. The limit on physical memory has been changed from 8 megabyte to 64 megabyte, and the limit of 15 mounted file systems has been changed to 255. The maximum file system size has been increased to 8 gigabyte, number of processes to 65536, and per process size to 64 megabyte of data and 64 megabyte of stack. Note that these are upper bounds, the default limits for these quantities are tuned for systems with 4-8 megabyte of physical memory.

3.1.6 Memory Management

The global clock page replacement algorithm used to have a single hand that was used both to mark and to reclaim memory. The first time that it encountered a page it would clear its reference bit. If the reference bit was still clear on its next pass across the page, it would reclaim the page. The use of a single hand does not work well with large physical memories as the time to complete a single revolution of the hand can take up to a minute or more. By the time the hand gets around to the marked pages, the information is usually no longer pertinent. During periods of sudden shortages, the page daemon will not be able to find any reclaimable pages until it has completed a full revolution. To alleviate this problem, the clock hand has been split into two separate hands. The front hand clears the reference bits, the back hand follows a constant number of pages behind reclaiming pages that still have cleared reference bits. While the code has been written to allow the distance between the hands to be varied, we have not found any

† Ethernet is a trademark of Xerox.

algorithms suitable for determining how to dynamically adjust this distance.

The configuration of the virtual memory system used to require a significant understanding of its operation to carry out simple tasks such as increasing the maximum process size. This process has been significantly improved so that the most common configuration parameters, such as the virtual memory sizes, can be specified using a single option in the configuration file[3]. Standard configurations support data and stack segments of 17, 33 and 64 megabytes.

3.1.7 Signals

The 4.2BSD signal implementation would push several words onto the normal run-time stack before switching to an alternate signal stack. The 4.3BSD implementation has been corrected so that the entire signal handler's state is now pushed onto the signal stack. Another limitation in the original signal implementation was that it used an undocumented system call to return from signals. Users could not write their own return from exceptions; 4.3BSD formally specifies the *sigreturn* system call.

Many existing programs depend on interrupted system calls. The restartable system call semantics of 4.2BSD signals caused many of these programs to break. To simplify porting of programs from inferior versions of UNIX the *sigvec* system call has been extended so that programmers may specify that system calls are not to be restarted after particular signals.

3.1.8 System Logging

A system logging facility has been added that sends kernel messages to the syslog daemon for logging in /usr/adm/messages and possibly for printing on the system console. The revised scheme for logging messages eliminates the time lag in updating the messages file, unifies the format of kernel messages, provides a finer granularity of control over the messages that get printed on the console, and eliminates the degradation in response during the printing of low-priority kernel messages. Recoverable system errors and common resource limitations are logged using this facility. Most system utilities such as init and login, have been modified to log errors to syslog rather than writing directly on the console.

3.1.9 Windows

The tty structure has been augmented to hold information about the size of an associated window or terminal. These sizes can be obtained by programs such as editors that want to know the size of the screen they are manipulating. When these sizes are changed, a new signal, SIGWINCH, is sent the current process group. The editors have been modified to catch this signal and reshape their view of the world, and the remote login program and server now cooperate to propagate window sizes and window size changes across a network. Other programs and libraries such as curses that need the width or height of the screen have been modified to use this facility as well.

3.1.10 Configuration of UNIBUS Devices

The UNIBUS configuration routines have been extended to allow auto-configuration of dedicated UNIBUS memory held by devices. The new routines simplify the configuration of memory-mapped devices and correct problems occurring on reset of the UNIBUS.

---

[3] The rumor that former Berkeley gurus were making significant incomes by consulting on how to increase the maximum process size is completely untrue.

3.1.11 Disk Recovery from Errors

The MASSBUS disk driver's error recovery routines have been fixed to retry before correcting ECC errors, support ECC on bad-sector replacements, and correctly attempt retries after earlier corrective actions in the same transfer. The error messages are more accurate.

3.2 Functional Extensions to Libraries and Utilities

Most of the changes to the utilities and libraries have been to allow them to handle a more general set of problems, or to handle the same set of problems more quickly.

3.2.1 Name Server

In 4.2BSD the name resolution routines (*gethostbyname*, *getservbyname*, etc.) were implemented by a set of database files maintained on the local machine. Inconsistencies or obsolescence in these files resulted in inaccessibility of hosts or services. In 4.3BSD these files may be replaced by a network name server that can insure a consistent view of the name space in a multimachine environment. This nameserver operates in accordance with Internet standards for service on the ARPANET [Mockapetris83].

3.2.2 System Management

A new utility, *rdist*, has been provided to assist system managers in keeping all their machines up to date with a consistent set of sources and binaries. A master set of sources may reside on a single central machine, or be distributed at (known) locations throughout the environment. New versions of *getty*, *init*, and *login* merge the functions of several files into a single place, and allow more flexibility in the startup of processes such as window managers.

A new utility keeps the time on a group of cooperating machines synchronized to within 30 milliseconds. It does its corrections using a new system call that changes the rate of time advance without stopping or reversing the system clock. It normally selects one machine to act as a master. If the master dies or is partitioned, a new master is elected. Other machines may participate in a purely slave role.

3.2.3 Routing

Many bugs in the routing daemon have been fixed; it is considerably more robust, and now understands how to properly deal with subnets and point-to-point networks. Its operation has been made more efficient by tuning with the use of execution profiles, along with inline expansion of common operations using the kernel's *inline* optimizer.

3.2.4 Compilers

The symbolic debugger *dbx* has had many new features added, and all the known bugs fixed. In addition *dbx* has been extended to work with the Pascal compiler. The fortran compiler *f77* has had numerous bugs fixed. The C compiler has been modified so that it can, optionally, generate single precision floating point instructions when operating on single precision variables.

4. Security Tightening

Since we do not wish to encourage rampant system cracking, we describe only briefly the changes made to enhance security.

4.1 Generic Kernel

Several loopholes in the process tracing facility have been corrected. Programs being traced may not be executed; executing programs may not be traced. Programs may not provide input to terminals to which they do not have read permission. The handling of process groups has been tightened to eliminate some problems. When a program attempts to change its process group, the system checks to see if the process with the pid of the process group was started by the same user. If it exists and was started by a different user the process group number change is denied.

4.2 Security Problems in Utilities

Setuid utilities no longer use the *popen* or *system* library routines. Access to the kernel's data structures through the kmem device is now restricted to programs that are set group id "kmem". Thus many programs that used to run with root privileges no longer need to do so. Access to disk devices is now controlled by an "operator" group id; this permission allows operators to function without being the super-user. Only users in group wheel can do "su root"; this restriction allows administrators to define a super-user access list. Numerous holes have been closed in the shell to prevent users from gaining privileges from set user id shell scripts, although use of such scripts is still highly discouraged on systems that are concerned about security.

5. References

[GADS85] GADS (Gateway Algorithms and Data Structures Task Force), "Toward an Internet Standard for Subnetting," RFC-940, Network Information Center, SRI International, April 1985.

[Lankford84] Jeffrey Lankford, "UNIX System V and 4BSD Performance," *Proceedings of the Salt Lake City Usenix Conference*, pp 228-236, June 1984.

[Leffler84] Sam Leffler, Mike Karels, and M. Kirk McKusick, "Measuring and Improving the Performance of 4.2BSD," *Proceedings of the Salt Lake City Usenix Conference*, pp 237-252, June 1984.

[Mockapetris83] Paul Mockapetris, "Domain Names — Implementation and Schedule," Network Information Center, SRI International, RFC-883, November 1983.

[Mogul84] Jeffrey Mogul, "Broadcasting Internet Datagrams," RFC-919, Network Information Center, SRI International, October 1984.

[Nagle84] John Nagle, "Congestion Control in IP/TCP Internetworks," RFC-896, Network Information Center, SRI International, January 1984.

[Walsh84] Robert Walsh and Robert Gurwitz, "Converting BBN TCP/IP to 4.2BSD," *Proceedings of the Salt Lake City Usenix Conference*, pp 52-61, June 1984.

Upas - a simpler approach to network mail

David L. Presotto (research!presotto)

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

*Upas** is a mail interface that routes messages between existing network specific mailers, users, and user mailboxes. It uses a regular expression based mini-language to convert mail addresses into the commands needed to route the mail to the intended destination. Upas is the mail interface for the Eighth Edition of the Unix† Time-Sharing System.

Introduction

Our entry in the 'mail race' sprang from events similar to those motivating the development of many mail systems. For many years a short and simple mailer was used to deliver local mail and to route mail via our home grown networks. Although its user interface left a little to be desired, its reliability was so high that great trust was put into it. However, as we gained access to more and more networks, particularly ones over which we had no control, the situation quickly deteriorated. Each of these networks had their own mail 'standards' and addressing conventions. With some trepidation, we absorbed these standards into our mailer. Its simplicity was quickly lost along with its fabled reliability. Realizing our danger, we decided to step back and see if there was a way to get back to a simple, well understood, and thereby reliable mail system.

The job to be performed by a network mail system is illustrated by figure 1. A mail system is essentially a large switch for handling the routing and delivery of messages. As a router it must be conversant in the various network protocols, be able to decipher destination addresses, and pass messages along to the next network. Sometimes it actually gets to deliver a piece of mail to a mailbox. Also, since there is no common mail format, the mail system must convert messages from one format to another as it routes them from network to network. Because of the number of networks and mail formats, this can easily lead to thousands of lines of code. Our task was to decide how to partition the task in order to create a manageable yet efficient mail system.

Some Observations

The task of interfacing to a particular network is often a messy and arbitrary thing. Fortunately, most entities (corporations, governments, committees) that design network protocols also provide code (i.e. mail programs) that understand these protocols. In our experience, it has always been easier to interface one of these mailers to our mail system than to incorporate the new protocols into our existing mailer. Also, code provided by someone else is supported by someone else. As network protocols change it is easier to pick up the new version of the network mailer than to rewrite our mailer.

Although there are many networks, there are far fewer message formats. It is clear that a message needs a destination address and possibly even a reply address. However, the imposition

---

* **upas,**[1] *u´pas, n.* (in full **u´pas-tree´**), a fabulous Javanese tree that poisoned everything for miles around; Javanese tree (*Antiaris toxicara*, of the mulberry family): the poison of its latex. [Malay, poison.]

† Unix is a trademark of AT&T Bell Laboratories

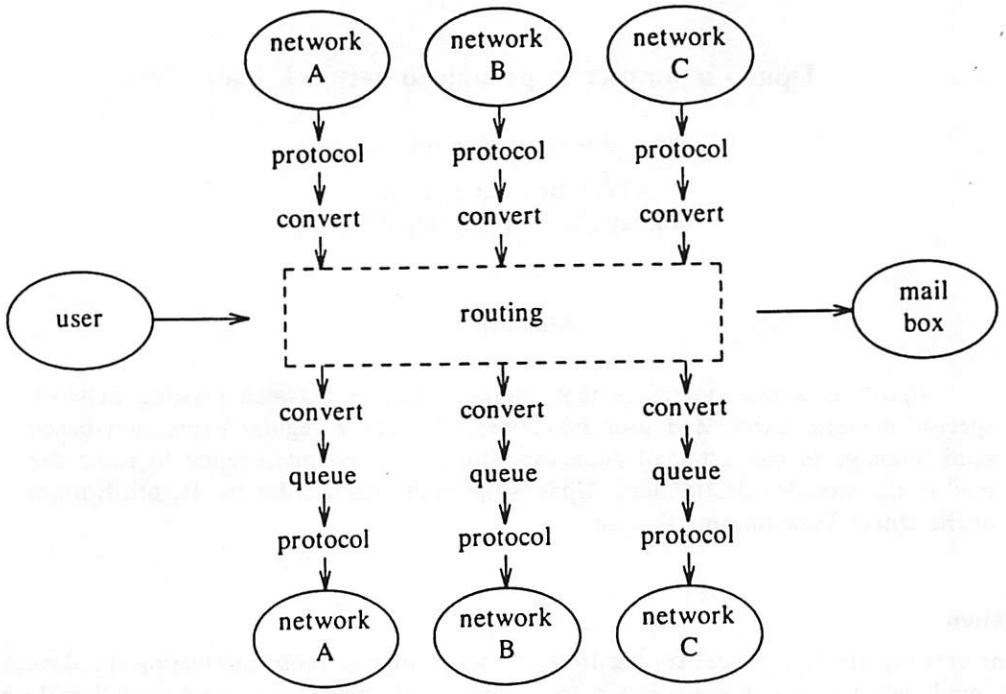


Figure 1. The functions to be performed to route network mail.

of further structure on the message is at best distasteful, at worst obstructive. Imagine what postal delivery would be like if the Postal Service opened each piece of mail to ensure that it is correctly dated and signed, that the form of address is correct, and that the company letterhead obeys some preconceived format, refusing delivery if any of these conditions are not met. Unfortunately, some networks impose such requirements. For a message to obey one standard is difficult enough. To expect it to survive a number of conversions between restrictive standards constitutes wishful thinking. Because of this, most networks adopt standards established by older or larger networks. Therefore, although there are many networks, there are relatively few message formats.

A network address describes a path through a number of machines and networks. This path may be rather simple, consisting of a single machine and user name. Often, however, the path crosses a number of administrative domains. Each such domain imposes some rules for structuring paths within the domain. Unfortunately, there is no adhered-to standard for binding the path segments from each domain into a single address. The networks differ on direction of binding (person@machine vs. machine!person), delimiters ('.' vs. '@' vs. '%'), quotation marks, and even case sensitivity. Therefore, there is no fixed way to correctly parse and understand a network address. Instead, there are conventions which tend to be very short lived, usually until someone issues a new RFC or a new network appears. As a relatively simple example, consider a message sent from one UUCP[2] network, through ARPAnet, to another UUCP network. The address format might be something like:

```
placeA!placeB!"placeD!placeE!person"@placeC
```

The rules for parsing such an address are easily defined. Unfortunately, the conventions underlying the rules change from day to day. Once you've managed to write your code, the administrator at placeB may decide that he won't accept quotation marks in an address and would really like the address to look like:

```
placeA!placeB!@placeC!placeD!placeE!person
```

A new set of parsing rules now have to be defined. In our experience these changes happen with maddening frequency. They are the direct result of there being no single comprehensive standard

or administrative authority. Therefore, we have to treat address parsing rules as ephemeral. Any network mailer should be able to change its address parsing rules frequently and with little difficulty. Tying them to one particular standard such as this week's ARPA rules is equivalent to planned obsolescence.

Finally, we should make a point about reversibility that many other mail designers seem to have missed. In addition to parsing destination addresses, mailers are expected to maintain some form of return address attached to the message. This often involves changing the current return address to one that the mailer will accept as a reply destination. A mailer should parse and modify return addresses using the same rules as it does for destination addresses. Otherwise, as is too often the case, the mailer will reject the very addresses that it has provided for replies.

A Solution

The best solution would have been to throw out all the so-called standards and create a single coherent scheme for formatting and addressing mail.[3] However, since we have no power to impose such a scheme, we have tried to use the above stated requirements and observations to build a mail system that makes the best of a bad situation.

The structure of our mail system is depicted in figure 2. Each network has its own interface program for message reception and transmission. In general these are the network specific mailers provided with the networks. When a message enters from a network, the network specific mailer gives it to Upas. Upas then either deposits the mail in a local mail box or routes the mail to the next network. A format specific filter may be called to convert the message from network format to one Upas understands or vice-versa; RFC 822[4] and Unix formats are built in. The rest of this paper describes how Upas preforms its message routing and the ease with which new networks can be added.

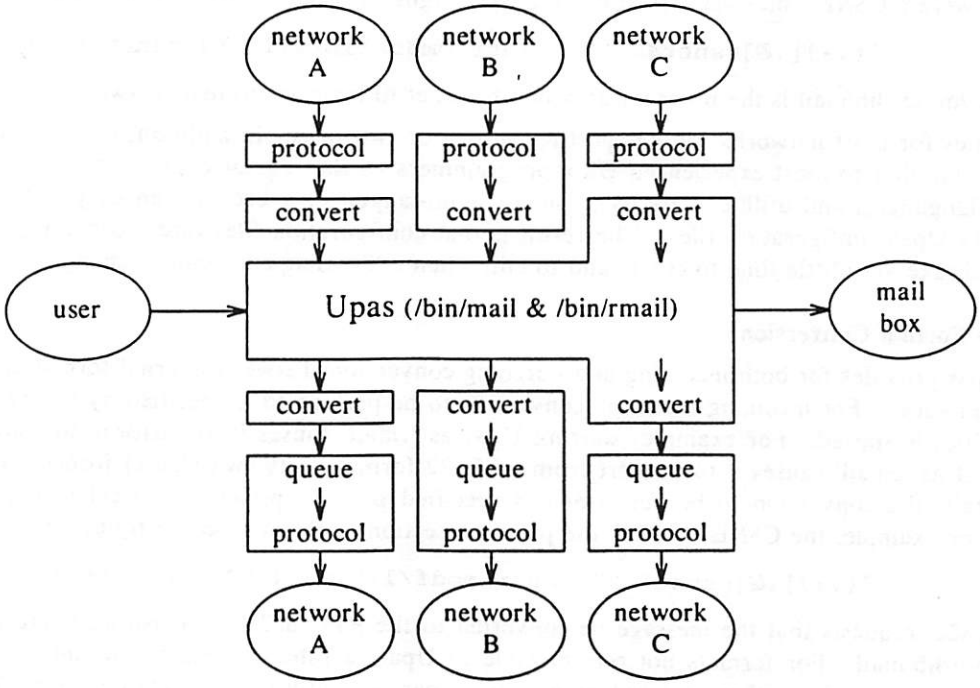


Figure 2. The structure of Upas. Each solid box represents a separate program.

Message Routing

The routing of messages is determined by a destination address and by a set of rewriting rules kept in a configuration file, /usr/lib/upas/rewrite. Each line of the file is a rule. Blank lines and lines beginning with '#' are ignored.

Each rewriting rule consists of four strings:

pattern An *ed*(1)-like regular expression, with simple parentheses () playing the role of \(\) and with the + and ? operators of *egrep*(1). The *pattern* is applied to mail addresses.

machine An *ed*(1) style replacement string that picks the source or destination string from an address matched by the *pattern*. The substring, '\s', is replaced by the login id of the sender.

command An *ed*(1) style replacement string to generate a command to deliver messages to the destination matched by the *pattern*.

conversion The name of a conversion to be performed.

When delivering a message, Upas starts with the first rule and continues down the list until a pattern matches the destination address. If the rule contains no command, the mail is appended to a local user's mailbox. If the rule does contain a command, Upas starts the command and pipes the message to it, performing any requested conversion. For example, the following rule can be used to specify that all RFC 822 style addresses should be passed to a particular gateway machine (research):

```
^[^!]+[.@%].+$  ""  "uux - research!rmail \\(&\\)"
```

The first expression, `^[^!]+[.@%].+$`, is used to match the destination address and the third, `"uux - research!rmail \\(&\\)"`, is used to form the command. The second expression (blank here) relates to forwarding restrictions and is explained below. The gateway machine can have more restrictive rewrite rules to pass a message onto a particular network. For example, if a message were a CSNET message, the following rule might exist on the gateway machine:

```
^(.+)[.@]csnet$  ""  "/usr/mmdf/lib/mail \1.csnet-relay"
```

Here /usr/mmdf/lib/mail is the program provided by csnet to interface to their network.

Rules for most networks can be specified in one or two lines. In addition, the rules are in a language familiar to most experienced Unix programmers — the regular expressions seen in many editors, languages, and utilities. By using such a mini-language, it becomes an easy task to build or modify Upas configuration files. The result is that configuration files rarely contain gross mistakes and take very little time to create and to edit when addressing conventions change.

Message Format Conversion

Upas provides for both incoming and outgoing conversions between internal format and those of the networks. For incoming mail, the conversion to be performed is specified by the name with which Upas is started. For example, starting Upas as 'rmail' causes it to perform no conversion. Starting it as 'cmail' causes it to convert from RFC822 format to its own (Unix) format. For outgoing mail, the conversion to be performed is specified by the optional last field in the rewrite rules. For example, the CSNET rule in the previous section might more correctly be written:

```
^(.+)[.@]csnet$ "" "/usr/mmdf/lib/mail \1.csnet-relay" rfc822
```

The 'rfc822' requests that the message be converted to the RFC 822 format before being piped to /usr/mmdf/lib/mail. For formats not recognizable to Upas, a filter process can be interspersed to perform the conversion either on input or output. Upas currently understands only RFC 822 and Unix message formats.

Discouraging Anti-social Behavior

It is often desirable to control the use of a machine as a forwarding site. Such is especially the case when the forwarding of mail has a devastating effect on either available cycles and telephone usage. The second field in a rewriting rule is used to allow only *friendly* machines to forward mail through Upas. This field, the *machine* field, is a replacement string matching the source and destination machine names from the reply and destination addresses. These names are checked against a file, /usr/lib/upas/forwardlist, which contains a list of friendly machines. If neither name is found in the list, the message is returned to the sender. For example, a UUCP address may have the following rewriting rule:

```
^([^!]+)!([^!]+)$    \1  "uux - -a \s \1!rmail \\(\2\\)"
```

Mail sent to "placeA!person" by "placeB!person" will be refused if neither placeA nor placeB is in the forwarding list.

If the file of friendly machines doesn't exist, the mail is forwarded unconditionally.

User Control

Users often wish to specify alternate ways to dispose of their mail. Upas allows this in a manner similar to Sendmail.[5] The first line of a user's mail file is interpreted as a command to the mail system. If the line is of the format

Forward to list-of-addresses

the mail is forwarded to each recipient in *list-of-addresses*. While this can be used to forward a single user's mail, it can be also be used to create mailing lists. To do this, one creates a file in the mail directory whose name is that of the mailing list and which consists of "Forward to" followed by the list of recipients.

If the first line is of the format is

Pipe to shell-command

shell-command is executed when mail is delivered, with the message as standard input.

Loop Detection

Detecting forward loops, like those provoked by 'Forward to' is difficult. It involves combining the forwarding lists of all involved machines into a single directed graph and then performing a search or partitioning to detect cycles. However, if we allow a detection algorithm to reject some legal although highly unlikely cases along with real loops, we greatly simplify the problem.

In the case of a single machine, an infinite forwarding loop corresponds to infinite recursion of the mailer. If a mailer rejects any message that results in recursion past a certain depth, it will reject all loops and some small number of legal but very long mail redirections. In our case a depth is 32 and to date, no legal forwarding loop has been more than 3 steps long.

In the case of a multi-machine loop, the recursion technique is not valid. However, we can still use a similar method. Instead of counting recursion, we scan the From line to see the number of times the local machine name occurs in the path. If this exceeds a limit (in our case 8), the mail is returned to the sender.

A Comparison With Sendmail

Upas is an attempt to solve the same problem previously attacked by Sendmail. Upas owes much of its design and success to Sendmail. The idea of designing Upas as a central switcher communicating with network specific mailers comes directly from Sendmail. The reasons we wrote Upas and didn't just adopt Sendmail are:

- We strongly favor messages whose only formatted portion are the destination and reply addresses. Sendmail has an unfortunate predilection for verbose and rigidly structured messages that we would like to avoid.

- Sendmail configuration files are famous for their inscrutability. We wanted a system that had simpler and therefore more easily verifiable rewriting rules.
- Sendmail is over designed for our needs including queuing (performed by our network specific mailers), SMTP support, batching of mail transmissions, aliasing, inclusion, etc. This extra design makes Sendmail more complicated and harder to support.

Summary

We have presented a simple yet flexible network mail system. It gains its simplicity from a number of assumptions which are valid in most networked computers. By using existing network specific mailers as expert systems that deal with network details, Upas itself remains relatively simple and understandable. By converting the most common message formats internally while allowing other formats to be converted by filter processes, Upas remains efficient yet extensible. Finally, by using a mini-language already familiar to most Unix programmers, Upas is easily modified to respond to changes in the name space and topology of the network.

References

1. *Chambers Twentieth Century Dictionary*, W & R Chambers Ltd., Edinburgh, GB (1976).
2. Nowitz, D. A., "Uucp Implementation Description," *Unix Programmer's Manual, Seventh Edition, Volume 2*, Bell Laboratories (October 1978).
3. Pike, R. and Weinberger, P. J., "The Hideous Name," *USENIX Summer Conference Proceedings* (June 1985).
4. Crocker, D. H., "Standard for the Format of Arpa Internet Text Messages," *RFC 822*, Menlo Park, California, Network Information Center, SRI International (August 1982).
5. Allman, Eric, "SENDMAIL - An Internetwork Mail Router," *Unix Programmer's Manual, 4.2 BSD, Volume 2C*, University of California, Berkeley (July, 1983).

SM: A SMALL MAILER

Eben Ostby, Allan Kaplan
Computer Division

Lucasfilm Ltd.
P.O. Box 2009
San Rafael, CA 94912

ABSTRACT

Electronic mail under the UNIX† operating system has always been designed around the program "/bin/mail", a simple program which puts things into a user's mailbox file and permits her to read the mail in the mailbox file. Inter-machine mail is accomplished via the uucp (UNIX —to— UNIX copy) utility. The advent of Berkeley UNIX 4.2 has seen many changes and additions to the mail system, including support for network mail (over tcp/ip Ethernet * links), DARPA Internet standard mail formats (RFC 819, RFC 821, and RFC 822), mail aliasing and forwarding, and unwieldy header lines. Unfortunately, the cost of all this has been high, in part because these capabilities have been added as layers around /bin/mail, in part because of certain design goals that were adopted by the authors of the new mail package. We have rewritten the mail package so as to streamline it. It's faster now.

Introduction

Perhaps the most heavily-used utility on the Lucasfilm UNIX time-sharing systems is the mail system. A small set of programs permit users to compose and transmit messages to their fellow users, read incoming messages, and manage some sort of file of saved messages. This package consists of both a front-end program, usually *Mail*[5], with which the mail user interacts, and a small set of programs which manage the delivery of mail messages.

The delivery subsystem is, in fact, responsible for more than just the delivery of a mail message to a user's mailbox file. It must *alias* the address list — that is, take each recipient name and replace that name with one or more names from an *alias database*. This step permits users to reside on various machines and yet retain a single, well-known address across a number of hosts. Aliasing may also be used to maintain mailing-lists. At Lucasfilm the mail system also acts as a front end for a number of programs. Messages may be "mailed" to a file, or the mailer may invoke a program with the text of a mail message as input.

The delivery subsystem is also responsible for the inter-machine transmission of mail messages. This may mean either transmission within a local area network, or connection to a remote site (in our case, via UUCP. [3])

† UNIX is a Trademark of Bell Laboratories.

* Ethernet is a trademark of Xerox Corporation

During early analysis of our 4.2BSD system, we found that the mail delivery subsystem contributed a great deal to the performance degredation we saw when we converted from our pre-4.2 release [2]. This provided incentive for us to rewrite the mail delivery program *Sendmail*, with beneficial results.

1. Sendmail

4.2BSD UNIX supports the *Sendmail* package [1], a mail transport program which parses mail addresses, speaks SMTP (Simple Mail Transfer Protocol)[4], handles address aliasing, figures out what program to call to deliver a mail message, and so forth. Sendmail, for some reason, is six or seven programs in one. Depending on how Sendmail is invoked, it will

- transmit a mail message,
- run as a SMTP daemon receiving mail,
- update the aliases database,
- print out the contents of the mail queue,
- clean out the mail queue,
- tell you how it would parse a given address,
- initialize its internal tables for you,
- try to send all the mail currently in the mail queue.

Not only does it do all these things reasonably well, but it is also *completely configurable*. This means that any action it can conceivably take is controllable by setting some flag or variable in a configuration file. The idea here is to minimize the chore of distributing and maintaining *Sendmail* in a multiple machine environment. Only one copy of the executable need be created for each machine type; host-dependent differences in the mailer can be controlled by the configuration file. Unfortunately, there are a number of drawbacks to this approach: the configuration file quickly becomes bigger than the program it configures (if one is not careful, anyway); the program must read the (substantial) configuration file each time it runs; the additional complexity makes for a slower, harder-to-maintain program; the systems staff must learn how to program the configuration file; yet the program must *still* be recompiled for each different machine and operating system on which it might run.

And, after all this, for most mail, *sendmail*, after doing a substantial amount of work, creating two or three new processes, ends up invoking /bin/mail. So, although the sendmail approach has its advantages, they may not outweigh the problems of its approach. That is to say, less is still probably more.

2. SM

Our approach in rewriting the mail transport system was to strip down the mail handler to the smallest possible set of functions. We believe that programs work best when they do as little as possible. There are a number of things that the mailer must do. In this case, these are:

- perform aliasing
- parse addresses
- exec "mailer" programs (eg, uucp)
- talk to other mailers via SMTP
- deliver to mailboxes
- deliver to programs and files, as specified in the alias file
- queue undeliverable mail for later delivery

You will note that there are a number of sendmail features missing from this list. In particular, the queue maintenance functions and alias database maintenance functions are missing. Also missing is the SMTP daemon. There is one added to the list (deliver to mailboxes.) The list

is still pretty long.

The remaining facilities of sendmail have been parceled out to other programs. Rebuilding the alias file is a single, fairly small program. The daemon role is made up of many of the *sm* modules, with a few extra thrown in; nonetheless, it is a different program.

As does *sendmail*, *sm* maintains a queue of mail messages which, for some reason, were not able to be delivered to their intended host. This queue is kept in a simple ASCII format; therefore, existing UNIX tools (incorporated into shell scripts) can be used to maintain this queue. Mail retransmission over a network is performed by a simple shell script invoked by *cron*. Cleaning the queue is likewise performed by a shell script, run once a day. A small *awk* program has been written to print a summary of the mail queue. Running this program when the queue gets too big generally produces enough information to alert the operator why the queue hasn't been cleaned up. (Invariably it is because a machine on the local network is down or not talking on the Ethernet.)

By distributing the mail delivery functions amongst a number of simple programs, the mailer is kept relatively small. The mailer is written in C, which is at least as comprehensible as a sendmail configuration file (and no such file need be re-read and interpreted every time the mailer runs).

3. Operation

This section briefly describes the way SM works. Most of the details have been omitted from this description. Of course, only the code tells the complete story.

3.1 SM proper

The *sm* program consists of four basic routines: an argument and name parser, a header and mail parser, a delivery loop, and a queuing routine.

3.1.1 Address parsing

First, the argument list to *sm* is parsed; this process sets various important variables, including *sm*'s idea of the name of the sender, where the mail is coming from, etc. Argument parsing also builds a list of recipient addresses which are *aliased* using the information in the file /usr/lib/aliases. Aliasing is a recursive process: each name is *parsed*, breaking it into two parts: a host name and a recipient name. If the destination host and the origination machine are the same, the recipient name is aliased again; when the input and the output to the aliasing process are the same, the aliasing stops.

Each name is stored in the *s-table*, a hashed name table. In storing the names, duplicate names are eliminated. The completed table is linked together for eventual traversal.

3.1.2 Mail text processing

The input mail to *sm* is read in and saved in a temporary file. In so doing, *sm* scans the header lines and builds some of the information it needs from them. In particular, it must

- count the number of "Recieved" lines
- break the various "From" lines up and save their contents
  (this is so *sm* can insert its own idea of the sender there)
- parse and add commas to the "To" lines (if the -t flag is set,
  this means getting the recipient list from here, too.)

When this is completed, *sm* will have built an image of the mail. A second image is prepared with UUCP-style addresses in the header, should UUCP transmission be indicated. These images can be copied verbatim into mailboxes or given to programs as input.

3.1.3 Delivery

The delivery routine is one huge switch statement. At address parsing time, note was made of the method of delivery for each address. The delivery routine follows the chain of addresses, and for each one, either

- flocks the mailbox using **flock**(2), and writes the file to the end
- sends the mail via **uux**
- writes the mail into the given file
- calls a program with the mail as standard input
- talks SMTP to another mailer across the Ethernet

In the SMTP case, *sm* chains together (on separate linked lists) all mail for a given destination host. The entire chain is transmitted with a single network connection.

If the mail fits in *sm*'s internal buffer, the program can write the mail directly from this buffer. Otherwise, it must read and write the mail from a temporary file. In our experience, over 90% of all mail messages fit in this (small) buffer. This one improvement speeds delivery to large lists of people considerably.

3.1.4 Queuing

The queuing or spooling mechanism in *sm* is quite simple. *Sm* attempts immediate transfer of any mail destined for a host with which there exists an SMTP connection. Should this transfer fail, the undeliverable mail is written into a file in the directory /usr/spool/mailq. A shell script to try to send the mail is written at the same time. The latter file contains code to remove itself. In particular, if this file is called '/usr/spool/mailq/x.100', the contents of the file might look like this:

```
rm -f /usr/spool/mailq/x.100;sm -f sender -S /usr/spool/mailq/d.100 root@host
```

where *sender* is the name of the original sender of the mail, and its intended recipient is *root* on the machine named *host*. Should the attempt to retransmit the mail fail, a new shell file is created. This process is repeated periodically until retransmission succeeds, at which time the mail is removed by *sm*.

Queue maintenance is handled by two programs, *cleanq* and *runq*, described below.

3.2 Daemon

The daemon program *smd* is substantially the same as *sm*. The differences lie in the input: rather than a command line parser, input comes from a routine that handles smtp connections.

3.3 Newaliases

The newaliases program is very straightforward. Very little parsing is done on the aliases file; rather, the database is created directly from the input /usr/lib/aliases file.

3.4 Cleanq and runq

As noted above, the queue created by *sm* and its daemon counterpart is quite simple. *sm* creates mail files and simple shell scripts which resend those files.

Thus the "resending" program, *runq* need consist only of a program that, at regular intervals, runs all the shell files in the queue directory. This runs under the auspices of cron. Following is the *runq* program in its entirety:

```
#!/bin/sh
cd /usr/spool/mailq
if ln index lock; then
        cat x.* ¦ sh
        rm lock
fi
```

The alert reader will note that the actual attempt at transmission of the mail is performed by the single line

```
cat x.* ¦ sh
```

The remainder of the script honors the locking protocol for files in this directory. Since the mail is not touched when retransmission is attempted, the job of expiring old undelivered mail can be handled by looking at the modification time on the file; a shell program *cleanq* using *find* to do this has been written; it too runs under the auspices of *cron*. It is somewhat more involved than *runq*, and has been relegated to an appendix.

4. UUCP support

SM supports the traditional form of UNIX inter-machine communication, UUCP. Mail destined for a UUCP host is passed via SMTP to the machine on the local network that maintains UUCP connections to the outside world. An edited version of the mail message, with a UUCP-compatible header, is passed as standard input to *uux*.

Selection of the gateway host (that machine which has UUCP connections to the outside world) is made using special aliases in the alias database. Aliases of the form

```
!uucphost:        localhost,localhost...
```

cause mail destined for the named *uucphost* to be routed to one of the named *localhosts* for transmission via UUCP. A default uucp host can be specified with the alias

```
!*:       localhost
```

5. Performance

SM is at least two or three times faster than the 4.2 release of *sendmail*. The speed gained is through:

- smaller code.
- no config file to read in.
- simpler address parsing.
- using flock to lock mailboxes, rather than hard links.
- buffering smallish mail messages and writing them, rather than rereading a temporary file.
- incorporating final delivery into *sm*.
- fewer *forks*.

6. Limitations

All this beauty comes at a price. *Sm* is not at all configurable. The code is written for an environment that supports UUCP and Ethernet links. The only address formats supported known to *sm* are

```
recipient
uucphost!recipient
recipient@etherhost
```

although *sm* also supports mailing to files and programs by aliasing a name to the file name or

"|programname", respectively.

Existing programs which use hard links to lock mailboxes must be modified to use flock. This change in the mailbox locking protocol considerably simplifies the locking of mailboxes, and makes the locks more reliable (since an *flock* lock is "removed" automatically when all references to it are closed).

Sm mails back error messages to users when it has trouble delivering mail. It tries to keep these notes fairly short; occasionally they become downright curt. *Sm* does not have as extensive error-checking as *sendmail*. *Sm* does not talk to the *syslog* error logging program, nor does it print messages on the system console. There is little in the way of debugging support. But then, it doesn't need much, does it?

7. Conclusions

We have described a reimplementation of the mail delivery subsystem. We believe it provides a useful object-lesson in the scaling of programs. That is, by carefully limiting the scope of the major programs we have designed a system that is significantly faster than its predecessor. We believe the reliance on small programs and existing UNIX tools has enhanced the comprehensibility of the package to programmers, as well. The *sm* system has been running at Lucasfilm and at other sites for over a year now, and has proved robust and flexible.

References

[1] Eric Allman, Sendmail Installation and Operation Guide. University of California, Berkeley. In Unix System Manager's Manual, 4.2 Distribution. March 1984.

[2] Sam Leffler, Mike Karels, and M. Kirk McKusick, "Measuring and Improving the Performance of 4.2BSD", *Proceedings of the Salt Lake City Usenix Conference*, pp 237-252, June 1984.

[3] Nowitz, D. A., and Lesk, M. E., *A Dial-Up Network of UNIX Systems.* Bell Laboratories. In UNIX Programmer's Manual, Seventh Edition, Volume 2. August, 1978.

[4] Jon Postel, Simple Mail Transfer Protocol. RFC 821. Network Information Center, SRI International, Menlo Park, California. August 1982.

[5] Shoens, K., *Mail Reference Manual.* University of California, Berkeley. In UNIX Programmer's Manual, Seventh Edition, Volume 2C. December 1979.

Appendix I - Using SM

SM is invoked by the mail front-end program, /usr/ucb/Mail. * When a mail message is created and ready to send, Mail execs a mail-delivery program; this may be *delivermail* (on pre-4.2 systems), *sendmail*, or *sm*. SM is invoked

sm [options] addresses...

SM takes the following command-line *options*:

* *sm* also works properly with other front-end programs, in particular, /bin/mail and the **mh** mail system. The latter requires the -t flag in *sm*.

—t	Obtain destination ("to") addresses from the "To:" and "Cc:" header lines, instead of from the command line arguments.
—m	Send mail to me, too, even if I am in an alias expansion.
—V	Print out the addresses the mail is being delivered to, as it delivers them.
—v	Read the mail and process as for —V, but don't deliver the mail (for debugging).
—f	The following argument represents the name of the sender of the mail.
—r	A synonym for —f.
—S	The following argument is the name of a file to be taken as the input mail. If the mail must be queued for later delivery, the file will not be touched; *sm* will arrange things such that the later queue runs will be given this same file. If the delivery succeeds, *sm* will remove (unlink) this file.
—d	Various debugging flags.

Setting up SM

A few files and such must be set up in order that *sm* run properly. Here they are:

Queue directory

SM maintains a directory of files for later delivery to hosts that are down, or to which communication is impossible at the moment. This directory must be created

```
mkdir /usr/spool/mailq
```

and a file containing the current file id number must be placed there:

```
echo -n 1 > /usr/spool/mailq/index
```

Cron entries

The following entries should be inserted in /usr/lib/crontab. They try to send to remote hosts and clean the queue, respectively. Both programs honor a lock in /usr/spool/mailq called 'lock'; it is a hard link to the file /usr/spool/mailq/index.

```
12,42 * * * * runq
26 23 * * * cleanq
```

RC entries

The *sm* daemon is usually started automatically at boot time from the script */etc/rc* or */etc/rc.local*.

```
if [ -f /usr/lib/smd ]; then
        (cd /usr/spool/mailq; rm -f lock; /usr/lib/smd) & echo -n ' sm'        >/dev/console
fi
```

Newaliases

The program *newaliases* rebuilds the aliases database, which provides reasonably fast access to the various mail aliases. It must be run in order for changes to /usr/lib/aliases to show through to *sm*. It may be run in one of a number of ways:

newaliases alone will rebuild the database

newaliases —a	will rebuild it only if it seems out-of-date. This is useful where newaliases is run regularly each night.

The —v flag may be given as well, to force newaliases to give you a summary (''verbose'') of the operation.

Appendix II - Cleanq

As promised, here is a small shell program which finds and removes mail which is deemed 'too old'. A letter is returned to the sender with a warning message and the text of the original mail.

```
#!/bin/sh
MAILBOY = "MAILER-CLEANER"
time = 3
cd /usr/spool/mailq
for TRIED in 1 2 3 4 5; do
        if ln index lock; then
for i in `find . -name 'd.*' -mtime + "$time" -print ¦sed 's".*/""`; do
        IFS = " "
        eval `fgrep "$i " x.* ¦ sed 's/.*sm -f//' ¦ awk '{ print "(echo '"'"'To: " $1
        print "Subject: undelivered mail"
        print ""
        printf "Your mail to:"; for (i = 4;i< = NF;i + + )printf " %s",$i; print ""
        print "was undelivered after '" $time "'days. It has been burned, the ashes scattered."
        print "---------- unsent mail ---------'"';cat $i) ¦ sm -f $MAILBOY "'" $1'}`
        IFS = "
"
        rm -f `fgrep -l "$i " x.*` $i
done
rm -f lock
exit 0
        else
                sleep 240
        fi
done
echo "To: Postmaster
Subject: Couldn't clean the queue

Cleanq didn't run as the queue remained locked for 20 minutes.
You may wish to investigate." ¦ sm Postmaster
```

Sendmail Revisited

Eric Allman

Britton Lee, Inc.
1919 Addison, Suite 105
Berkeley CA 94704

Miriam Amos

Digital Equipment Corporation

Computer Systems Research Group
University of California
Berkeley CA 94720

Sendmail, the internetwork mail router released on the 4.2 BSD UNIX distribution, has been available for over two years. During that time we have heard many ardent arguments both for and against *sendmail*. This paper looks retrospectively at *sendmail*, discussing both the successes and the failures of this approach.

Sendmail implements a general internetwork mail routing facility, featuring aliasing and forwarding, automatic routing to network gateways, and flexible configuration. It neither interfaces with the user nor does the actual delivery. It simply collects the message generated by a user interface or an intermediate mailing channel, performs the necessary header manipulation required by the destination, and passes the message on. For a more complete description, see [Allman83].

Section 1 briefly describes the environment in which *sendmail* was developed. Sections 2, 3, and 4 describe our perceptions of *sendmail*'s most important successes, mixed successes, and failures respectively. Section 5 describes quirks in the mail environment that caused complications.

1. HISTORY

Early network mail systems usually presumed the existence of only one network. UUCP [Nowitz78], BerkNet [Schmidt79], and the ARPANET were significant examples of UNIX-based networks of this type. Most of them resorted to directly modifying the *mail* program to trap addresses containing the network character — such as '!', ':', or '@' used by the above examples.

When networks began to contact each other this approach suddenly developed problems. Some user interfaces could not handle all types of addresses. Incoming network mail could not always be forwarded to the correct gateway machine. The format of messages was different in different networks. At Berkeley the situation became critical when three networks were connected: UUCP on *ucbvax* and ARPANET on the INGRES 11/70, with BerkNet tying these and the other Berkeley machines together.

The *delivermail* program was developed at Berkeley to deal with the interconnection problem. Instead of ad hoc hacks to the mail system, routing was centralized in a single table. The rules were simple, based completely on the network characters. Heuristics were applied to disambiguate addresses with mixed separator characters, such as "a!b@c." Addresses that were determined to be destined for networks with a link on another host were

routed to a designated relay host. Information about the network configuration was compiled into the code. *Delivermail* did no direct delivery, but invoked programs to perform the interface to networks or local mailers. *Delivermail* also supported aliasing.

While *delivermail* provided some solutions to the changing mail environment, several deficiencies were quickly apparent. The compiled-in configuration complicated the installation of new versions of *delivermail* on the growing number of systems running the code. Multiple UUCP gateways were not possible. Messages passed through Berkeley to the ARPANET did not always obey the ARPA format standards defined by RFC 733 [RFC733]. No mail transport had yet been developed for the newly installed ethernet, which was ultimately intended to replace BerkNet.

This period was also one of rapid change in the mail environment. The ARPANET was undergoing a major upheaval in protocols to accommodate the rapid network growth that had been experienced (from under 200 hosts to many thousands of hosts in a few years). DARPA (the Defense Advanced Research Projects Agency) had funded Berkeley to provide a version of UNIX that would support the new ARPANET protocols.

These demands lead to another evolution of the mail system. The configuration information was moved out of the code to a file that was read at startup time, thus reducing the difficulty of installation. The address parsing was changed by the introduction of *rewriting rules*. The rewriting rules modified text using a simple pattern match/replacement algorithm. The same rewriting rules could also rewrite a message header so that it would obey ARPANET protocols. Support was added for the ethernet. The resulting program was dubbed *sendmail*.

2. SUCCESSES

In many ways *sendmail* has been a phenomenal success. We believe strongly that developers of future mail systems should insure that these features are propagated.

2.1. User Interface/Transport Dichotomy

Separating the user interface from the transport and delivery services has proven invaluable. It corresponds to the "one tool for one job" concept[1], allows multiple front ends, and helps insure consistent addressing schemes.

Mailers can be implemented independently of other mailers in the system, much as UNIX device drivers are independent of other device drivers. Since the mailers obey normal UNIX conventions for process invocation, they can be debugged before integration into the mail system.

By centralizing addressing, such important features as aliasing are automatically available to all users. Previously */bin/mail* allowed no aliasing, the UCB *Mail* [Schoens83] program had one alias file, MH another, and incoming ARPANET mail a third. Maintaining consistent name spaces was essentially impossible in that environment.

2.2. Reliability

A prime implementation consideration was reliability, in the sense that any message that has been accepted should be delivered, returned, or passed to a human who can take appropriate action, even in the presence of system crashes, network outages, etc. In particular:

---

[1]It has been observed that one of the great successes of UNIX is that each tool does only one job, and therefore can do that job well. For example, the *sort* program sorts, without converting data, rearranging fields, adding headers and footers, summing columns, etc.

- Messages that cannot be delivered should produce a notification to the sender, assuming a legal sender can be determined.
- If notification cannot be returned, that notification should be delivered to a central "postmaster" for human intervention.
- If the central postmaster cannot be contacted (usually due to a trashed alias file), send the response to *root*.
- If root cannot be contacted (usually due to a trashed */etc/passwd* file), put the response in a known place and log a catastrophic failure.

A major component of reliability is the concept of *responsibility*. For example, before *sendmail* will accept a message (by returning exit status or sending a response code) it insures that all information needed to deliver that message is forced out to the disk. In this way *sendmail* has "accepted responsibility" for delivery of the message (or notification of failure). If the message is lost prior to acceptance, it is the "fault" of the sender; if lost after acceptance, it is the "fault" of the receiving *sendmail*.

This algorithm implies that a window exists where both sender and receiver believe that they are "responsible" for this message. If a failure occurs during this window then two copies of the message will be delivered. This is normally not a catastrophic event, and is far superior to losing a message.

2.3. Virtual Users

The ability to allow "virtual" users as recipients appears to be important. In sendmail these fall into three classes: aliases, files, and programs.

Aliases associate names with lists of addresses. Although aliases are well known, prior mail systems have often considered this a user interface function. The disadvantage of this is that different namespaces are associated with different user interfaces, and incoming network mail has no access to aliases at all.

Files allow mail to be stored in a named file on the system. This is normally used for logging long-term transcripts of group discussions.

Programs as recipients allows "active recipients." For example, the CSNET name server [Solomon81] permits automatic queries and updates by sending a message in a special format to a specified address. Other programs exist to submit mail to bulletin boards, return messages to the sender (for "I am away from my desk"-style notifications), and batch messages together for digest-style retransmission.

2.4. Consistency

Before *delivermail* was in place throughout Berkeley, each user interface interpreted addresses themselves. Not all interfaces handled all addresses in the same way (occasionally they couldn't handle some forms of addresses at all!), and almost all of them had different alias files.

Address interpretation and aliasing should be focussed in one module so that uniformity can be guaranteed. Furthermore, uniformity should be guaranteed across machines in a cluster; forwarding to a gateway machine should always be implicit[2].

[2]This is also an argument for distributed alias files. These have substantial implementation problems, especially as the cluster grows large. Grapevine [Birrell82] is probably the best current published example of a system offering true consistency in a distributed environment.

3. MIXED SUCCESSES

Items in this section have been both successes and failures. We attempt to describe the ambiguities of these features.

3.1. Rewriting Rules

Sendmail uses "rewriting rules" to perform address modification and parsing. Each individual rule (or production) rewrites a series of symbols (words, operators, or control information) into another form. The syntax is straightforward, with similarities to regular expressions.

However, in practice the rules can be arcane, much as a large *awk*(1) script can seem unapproachable. Also, novice maintainers are often confused about the order in which individual rulesets are invoked.

The very positive aspect of the rewriting rules is the power that they provide. This ability to resolve one address syntax and to rewrite the address into another syntax provides a gateway-like interface between heterogeneous mailing systems. The unlimited flexibility allows for future growth. It does not restrict the mail environment and does not restrict *sendmail* to the currently known environment. As the electronic mail environment continues to evolve the rewriting rules can accommodate its evolution.

For this reason we believe in the concept, although the current implementation needs improvement.

The alternative to rewriting rules appears to be a grammar. This was not used because there seems to be too much ambiguity in "the real world" — that is, when multiple address formats are accepted. An example of ambiguous parsing is the address:

a!b!u@c

which can parse differently depending on the values of *a*, *b*, and *c*. Making host names keywords so that a grammar can distinguish them is usually impractical.

3.2. SMTP in Sendmail

SMTP (the Simple Mail Transfer Protocol, defined by RFC 821 [RFC821]), controls transmission of mail over the ARPANET. It allows multiple recipients for each message, with fine control over status of individual recipients. SMTP is implemented as an integral part of *sendmail*.

Including SMTP as part of *sendmail* can be argued to violate the "one tool for one job" philosophy. Conceptually SMTP is another mail protocol that could have been implemented outside of *sendmail*. For example, a DECnet mailer would certainly be implemented as a separate program called by *sendmail*.

However, SMTP has many valuable features that are impossible to simulate using the *exec/wait* paradigm. For example, SMTP allows status checking of each individual recipient, while still permitting message batching. These features are important enough to deserve direct implementation.

We also believed that traffic would be sufficiently high over high speed local networks to merit including SMTP as a "backbone" protocol directly in *sendmail*.

3.3. Queuing

When SMTP was added, it became necessary to add a queue that would hold messages that could not be delivered immediately because the receiving host was unavailable. All of our previously available network interfaces included queuing, so this had not been an issue.

Including a queue seems to have several advantages. First, reliability can be improved, since a "checkpoint" of a message (in the form of the queue entry) can be saved fairly quickly. Second, responsiveness can be improved by allowing user interfaces to return as soon as the job is checkpointed. Third, new mailers can be simpler to write since they need not implement a queue themselves.

Implementing a queue requires a substantial amount of code. Reliability adds additional constraints: at no time can the information necessary to complete delivery of a message be deleted during a status update.

The ultimate decision seems to be philosophical: *delivermail* (without a queue) was a crossbar switch with no long term state, while *sendmail* is a store-and-forward facility. In general we believe that the advantages outweigh the problems.

3.4. Header Munging

In a sane world, header munging (modifying the text of the message header, such as To: lines) would be unpardonable. The process is extremely difficult to do correctly, and occasionally has the effect of producing unintelligible headers.

However, this "sane world" would include absolute addressing everywhere. For example, the address "host!user" would be interpreted identically on all hosts in the internetwork. Unfortunately the UUCP route-based syntax preempts this as a possibility, since "host" is interpreted relative to the current location. Substantial progress has been made in this area, but these new facilities are not yet universally available.

Our mythical "sane world" would also have consistent addressing syntaxes. Most ARPANET sites are unwilling to build in knowledge of UUCP-style addresses, British JNT addresses (which reverse the order of domain names), DECnet addresses (using two colons to separate node names), etc.

As a major gateway between different networks, Berkeley was required to insure messages released to a particular network obeyed the standards of that network. The choices seemed to be to reject nonconforming messages or to edit them into correct format. The former choice did not seem realistic, so header munging became inevitable. *Sendmail* was required to perform header munging.

The UUCP header munging could have been placed in the incoming UUCP module. This was rejected because of other sources of nonstandard headers, such as BerkNet and local user interfaces.

In practice, we know of no sites able to totally avoid some degree of header munging.

3.5. Adding Headers

In many cases *sendmail* will add lines to the message header. Augmenting the header differs substantially from modifying existing header lines.

In the "purest" UNIX sense messages require no header at all beyond the timestamp line that */bin/mail* adds. *Sendmail* rejects this totally for two reasons. First, we believe that mail must be more structured when users must process more than a few messages a day. This level is reached rapidly in a network environment. Second, Berkeley's DARPA funding mandated support of ARPANET mail protocols.

Header lines such as "Date:" and "From:" are required by ARPANET standards. "From:" also permits inclusion of a full name (rather than a login name); this becomes useful when the user population becomes large enough that the correspondence between login name and real name is not obvious; networks are automatically large enough to satisfy this criteria.

"Received:" lines help trace the message in the event of difficulties. These can help identify network congestion, misrouting of messages, etc. However, it seems quite clear that user interface agents should not normally display these lines.

"Message-Id:" lines are intended to assist a new generation of user interface agents that will provide better global semantics. For example, they could maintain a "reply chain" for ongoing discussions. Some work is beginning in this field. [Comer84] Of course, these lines should also normally be elided when displayed to a user under normal circumstances.

Clearly, there is a need for better user interfaces in the electronic messaging area. At a simplistic level they should delete header lines intended for machine interpretation or debugging; this feature is already available in programs such as Berkeley *Mail* [Schoens83] and *MH*. Ultimately they should maintain global semantics, permit multimedia mail, etc.

4. FAILURES

Sendmail has not proven completely adequate in many ways. In some cases these were design failures, in other cases implementation errors. This section describes some of our perceptions of *sendmail* failures, including our current feelings for how they might be avoided.

4.1. Not UNIX-like

Delivermail was designed as a "crossbar switch" for mail messages. A single program was reasonable in this case; the size was constrained, and the functionality reasonably limited. All mail was processed to completion in a single transaction.

Sendmail grew out of *delivermail* one feature at a time. It always seemed easier to add these features within the existing structure. At some point it became clear that *sendmail* had become a monolithic mess.

In retrospect, it seems clear that the addition of queuing was the critical point where *sendmail* violated the "one tool for one job" rule. The queue provides a natural interface between two distinctly different functions: adding a message to the queue and processing those messages. The queue also prescribes several ancillary functions: printing the queue, purging old jobs, etc.

Today, we would define these functions as separate programs:

- *Queueup* — insert messages in the mail queue. Called by the user, SMTP server, and cleanqueue. It in turn would call runqueue.
- *Runqueue* — process queue entries, taking an optional job name list to run only those jobs. Called by queueup or *cron*(8) to push the queue.
- *Printqueue* — print out the mail queue.
- *Cleanqueue* — return expired messages and do basic consistency checking.
- *Rmqueue* — remove a job from the queue.
- *SMTP Server* — process SMTP connections.
- *Newaliases* — build the alias file.
- *Editalias* — a front end to modify the alias database, providing security checks on modification rights (much like Grapevine [Birrell82]) and locking.
- *Verifyaddr* — verify an address list with varying levels of information. This is the same as /usr/lib/sendmail —bv.
- *Addrtest* — test the rewriting rules in the sendmail configuration file. This is the same as /usr/lib/sendmail —bt.

Not all of them are available today because of the difficulty of merging them into the existing structure.

4.2. Cryptic Configuration

The *sendmail* configuration file is noted for being cryptic. There seem to be two causes of this:

- The syntax is arcane. Although obscure syntaxes are not inherently evil (consider for example the */etc/passwd* file), it seems clear that when a file exceeds a certain size a more friendly syntax is required. The */etc/termcap* file survives only because entries may be viewed semantically independently from all other entries, although in other respects it suffers from the same disease.
- Excessive size/flexibility. The amount of power that the configuration file exerts is more than sufficient to completely mutilate mail beyond recognition or recovery. This power is implicitly confusing.

As a result of these problems *sendmail* is extremely difficult to install and operate.

Although a runtime configuration file appears to be an advantage, it seems clear that it should be small and clear. Appropriate changes seem to be:

- Use a cleaner syntax. This might require a *yacc*(1) grammar. If this was too slow to allow at runtime, a compiled form could be devised.
- Reduce the size of the configuration file. This is important both to simplify human understandability and to decrease startup time. A huge amount of the configuration file is "boilerplate" that could be compiled in.
- Use rewriting rules to convert from and to external formats only. Internally, use a hard-coded parse over a well defined form. This would ultimately be faster and have better error recovery.

4.3. Second System Effect

Sendmail suffers from second system effect [Brooks75]. It embodies too many functions. There appear to be several reasons for this.

First, in many cases requirements were changing or inconsistent, mandating unusual functionality. For example, *sendmail* heuristically determines whether spaces between words in a To: line separate addresses (as with Berkeley *Mail*) or are legal address components (as with RFC733 standard mail). This feature was easier to add than to try to change all copies of *Mail*, but still appears to be inconsistent with the primary job of delivering a message.

Second, *sendmail* made too much of an effort to be "all things to all people." For example, creating */bin/mail*-style "From" lines is purely a concession to UUCP. Building this knowledge into *sendmail* is an inappropriate violation of information hiding.

5. PROBLEMS IN THE ENVIRONMENT

In some cases problems in the "givens" of the mail environment created problems that might not otherwise have occurred. Although these are generally irrevocable and certainly out of our control, their description may prove interesting.

5.1. UUCP Related Problems

UUCP presented several interesting problems. First, the UUCP-style addresses were left-to-right associative, which conflicted with right-to-left associative ARPANET addresses. For example, the address:

a!b!u

specifies sending to host *a*, then to host *b*, then deliver to user *u*. However, the address

a!b!u@c

is ambiguous. Although techniques are known to resolve such ambiguities [Honeyman85] they require substantial knowledge of the topology of the internet.

Second, route-based addresses require modification every time they are forwarded. For example, the address "ukc!pc" is completely reasonable from many European sites, but is unknown from almost all American sites. Addresses of this form (both sender and recipient addresses) must be modified as transferred to remain usable in the target environment.

5.2. ARPANET-Related Problems

In January 1981 the ARPANET converted to new mail protocols (SMTP [RFC821] and RFC822 [RFC822]). On the whole the changes were excellent, with the following exceptions:

5.2.1. Route-addresses

The new ARPANET protocols accurately saw the need for explicit routing on occasion — that is, the ability to force a message to a particular site at which the addresses are reevaluated. Unfortunately the syntax used is difficult in two respects: it combines left-to-right with right-to-left address evaluation (albeit unambiguously), and it overloads the comma and colon operators. For example, to force a message through *hosta* and *hostb* before delivery to *hostc*, the syntax is:

⟨@hosta,@hostb:user@hostc⟩

Normally comma separates addresses. However, commas have different semantics when enclosed in angle brackets. Similar comments apply to the colon, which is normally used to indicate mailing lists.

An alternative syntax that would not overload operators, would be compatible with prior protocols, and which would preserve strict right-to-left parsing would be:

user@hostc@hostb@hosta

5.3. Changing Requirements

Sendmail evolved in a constantly changing world. Notably, while *sendmail* was being implemented the ARPANET mail protocols RFC821 and RFC822 where under development. Requirements were modified many times during development.

Changing requirements are still a reality; for example, as name servers become available (e.g., [Terry84]) the mail system will have to adjust.

6. SUMMARY

Sendmail tried to maintain old interfaces and add new interfaces, while simultaneously trying to keep up with changing protocols. These goals led to an exceptionally large, flexible implementation, rather than a smaller implementation that would be possible given a simpler environment. Despite these problems, *sendmail* has been quite successful in many ways.

Sendmail's primary successes have been reliability and consistency. These goals become especially difficult to insure in an internetwork environment; at the same time they become more important because of the increased difficulty of locating problems.

Sendmail's primary failures have resulted from failure to decompose the program into semantically consistent pieces and an overeagerness to please resulting in a form of Second

System Effect. These have resulted in a program that is far larger than appropriate for the amount of work it actually has to perform.

In some sense the problem of interconnecting existing electronic mail systems is impossible to solve without an overhaul of existing systems. Besides ARPANET and UUCP, *sendmail* has been connected to BITNET, DECnet, CSNET, ACSnet, and undoubtedly many others; connections to public carriers such as MCImail and TELEX have been discussed. The number of different message and address formats in these systems is staggering.

Overall we feel that although *sendmail* could easily benefit from reimplementation, most of the basic concepts currently implemented are appropriate and should be present in new systems.

REFERENCES

[Allman83] Allman, E., "Sendmail — An Internetwork Mail Router." In [UCB83], volume 2C. July 1983.

[Birrell82] Birrell, A. D., Levin, R., Needham, R. M., and Schroeder, M.D., "Grapevine: An Exercise in Distributed Computing." In *Comm. A.C.M. 25,* 4. April 1982.

[Brooks75] Brooks, F. P., *The Mythical Man Month — Essays on Software Engineering.* Addison-Wesley. 1975.

[Comer84] Comer, D. E., and Peterson, L. L., *Conversation-Based Mail.* Tilde Report CSD-TR-465. Purdue University, Computer Science Department. March 1984.

[Honeyman85] Honeyman, P., and Parseghian, P. E., "A Parser for Electronic Mail Addresses." In *Proc. of the Winter 1985 USENIX Conference.* Dallas, Texas. January 1985.

[Nowitz78] Nowitz, D. A., and Lesk, M. E., "A Dial-Up Network of UNIX Systems." In [UCB83], volume 2B.

[RFC733] Crocker, D. H., Vittal, J. J., Pogran, K. T., and Henderson, D. A. Jr., *Standard for the Format of ARPA Network Text Messages.* RFC 733, NIC 41952. Network Information Center, SRI International, Menlo Park, California. November 1977.

[RFC821] Postel, J. B., *Simple Mail Transfer Protocol.* RFC 821. Network Information Center, SRI International, Menlo Park, California. August 1982.

[RFC822] Crocker, D. H., *Standard for the Format of Arpa Internet Text Messages.* RFC 822. Network Information Center, SRI International, Menlo Park, California. August 1982.

[Schmidt79] Schmidt, E., "An Introduction to the Berkeley Network." University of California, Berkeley. 1979.

[Shoens83] Shoens, K. (revised by Leres, C.), "Mail Reference Manual (revised)." In [UCB83], volume 2C. July 1983.

[Solomon81] Solomon, M., Ladweber, L., and Neuhengen, D., *The Design of the CSNET Name Server.* CS-DN-2, University of Wisconsin, Madison. November 1981.

[Terry84] Terry, D. B., Painter, M., Riggle, D. W., and Zhou, S., *The Berkeley Internet Name Domain Server.* UCB/CSD 84/182. University of California, Berkeley, Computer Science Division. May 1984.

[UCB83] *UNIX Programmer's Manual,* 4.2 Berkeley Software Distribution. University of California, Berkeley. August 1983. Derived from *UNIX Programmer's Manual, Seventh Edition.* Bell Laboratories, August 1978.

All the Chips that Fit

Tom Lyon
Joseph Skudlarek

Sun Microsystems, Inc.

ABSTRACT

In this paper we discuss, from a software person's point of view, the problems found in employing complex VLSI chips in UNIX† systems, especially low cost workstation implementations. We give specific examples of clashes between the UNIX world and the chip world. We then discuss what attributes UNIX needs in chips, attributes we hope to see in future chips.

Trends

Workstations are expected to be small, full featured, fast, and affordable. Workstation manufacturers often meet these demands by composing systems with high density subsystems. Since board real estate is so valuable, there is a push to implement the functionality and performance of these subsystems with fewer and fewer chips. In many cases, the subsystems consist of one or a few VLSI chips and, when required, glue chips.

Chip manufacturers, on the other hand, are vying for sales in a crowded market. They can offer price, availability, and/or features. By offering a chip with many features, they may widen the market audience by encompassing more applications; thus they can sell more chips and amortize the development cost over more units, reducing the per-chip cost. They can also offer an advantage by providing functionality or performance with a small number of chips; for the chip user, this conserves board space, reduces manufacturing costs, and may reduce cost of goods.

So from both ends of the manufacturing spectrum, there is a desire for "all the chips that fit". But what is often neglected is the cost of actually employing the complex chips in a complete system, including the software development costs, and the memory costs to run the software.

The Bugs Are Coming

In the rush to get to market, chip vendors may make the software developer's life difficult. Vendors concentrate on securing design-ins by specifying attractive features or performance; board designers must often commit to the chip before production units are available. When the chips arrive, the users find that many of the specified features do not yet work. At this stage, the hardware design is often frozen, and it is the software people who must obtain and scrutinize the vendor's bug list for the chip, often available under clever names such as "errata sheet", "news", or "advanced topics". The bug list itself is often out of date, sometimes resulting in a "Yes, we know about that" reply from the chip's manufacturing or product engineer when you're explaining the bug that you have been hunting for three days and two nights. Many systems may have to be shipped before all the bugs in the chip are fixed (the norm for the delay between initial chip introduction and correct production chips is becoming 1 year or greater); so effectively the software is stuck living with the bugs for all time.

† UNIX is a Trademark of Bell Laboratories.

An example is the Intel 82586 LAN Co-processor. In addition to Ethernet, it was designed to support almost any other type of CSMA/CD LAN. Features include speeds from 1 to 10 Mbps, bitstuffing framing, 1 to 6 byte addresses, 16 or 32 bit CRC, etc. It relies heavily on internal microcode to handle all this, and, as in most complex software systems, the chip has bugs. The driver for this chip has been through several major revisions of buffer management to cope with bugs in the chip. Although the 82586 is full of features, the driver for the 82586 is the largest in our system, weighing in at more than 45 pages. There is a lesson here: Small is Beautiful.

Small is Beautiful

The mere presence of useless features can be detrimental because the programmer can easily select an incorrect mode and not notice it. For a while, the Sun driver for the Zilog Z8530 SCC (UART) was setting the chip to expect input characters of 8 data bits AND a parity bit AND to ignore that parity bit, instead of 8 data bits and no parity bit at all. We know of no other systems which use this combination. The result is that input characters would not be recognized unless there was at least one idle bit between them; this is certainly the case for human typing, and often the case for computer output. It was noticed however, that function keys on only some terminals were usually garbled, because they sent multiple characters with no intervening idle bits.

Allow Me!

Another problem in chip design is the tendency to put functionality in the chip which could be done just as well, or better, in the software driving the chip. In fact, users must sometimes program around incomplete solutions to create the required complete solution; this adds to the complexity of the code, costing in development time, and execution time and space. An example here is the National Semiconductor MM58167 Time-Of-Year Clock. This is a low power clock used to keep the system time when the system is powered off. The chip is apparently designed to appeal to users of microprocessors which can't do division and multiplication, for the chip keeps the time in separate registers for the month (no year), day, hour, minute, second, etc., rather than just providing a counter which counts at a known rate. In addition, each register is kept in BCD format, which makes displaying the time very easy. Unfortunately, UNIX doesn't want to display the time, it just wants something from which to set its 32 bit counter of seconds since Jan. 1 1970, GMT. An attempt to do a straightforward conversion from UNIX time to chip time fails because the chip has no support for leap years or daylight savings time. Ultimately, we ended up using the chip just as a bizarre series of modulo counters; the amount of arithmetic to convert chip time to UNIX time is disgusting.

The opposite problem is chips which require the software to specify operating modes when only one mode works in a system, and others may cause physical damage! An example here is hardware that requires software specification of interrupt vectors — this is convenient but a major disaster if done incorrectly. Yet software specification of modes is becoming the norm for complex chips because using extra pins on the chips to specify modes is prohibitively expensive.

On Holy Wars

Chips vendors want to sell their peripheral chips to a broad market, but they also want to encourage the market to use their CPU chips. To this end, the peripheral chips are often designed or marketed as being compatible only with that vendor's CPU. For instance, the Intel 82586 only supports Intel/DEC byte order; the Sun hardware swaps the bytes in order to get the data in the right order, but as a result any 16 bit control fields are swapped and must be dealt with in (you guessed it!) software.

Sometimes nomenclature is the problem. In one prototype design, a TI chip was used. It was discovered, by the software writers, that the bits in each data byte traveling to the chip was reversed; TI numbers their bits with bit 0 as the high-order instead of the low-order bit.

In one case, Sun had obtained early specifications for a certain chip and was intending to use it in a CPU board design. When the chip vendor learned that their chip was to be used with a another vendor's CPU chip, they revised the specification of the chip to make it LESS CLEAR whether or not this was possible!

Address Spaces

Microprocessor systems often have many distinct pools of memory and many different masters (processors or DMA devices). Not all memory pools can be accessed from every master, and often the memory which is shared has a different address for each master. These problems arise because not all masters support the same number of address lines (16, 20, 24, and 32 bit addressing is often found in a single system), and because memory bandwidth and the high cost of caches make true shared memory either slow or expensive.

The need to transform addresses from one master to another or to copy or map memory from one pool to another introduces significant overhead at low levels in the system. Example: the Intel 8237 is a 16 bit address DMA chip which is common on boards which plug into the Multibus, a bus with 24 bits of address, which is restricted to 20 bits in the Sun-2/120, of which 256K (18 bits) is a port into processor virtual memory (24 bits) which is mapped to the real address bus (22 bits). A "raw" read or write must transform the user address into a DMA port address into a Multibus address into a board address for the high 8 bits and a 8237 address (Intel byte order) for the low 16 bits. All this must be done twice if the request spans a 64K boundary. This is one of the simpler cases; we won't even attempt to describe doing disk transfers directly to and from the memory on the Sun-2 Multibus Ethernet card. Understanding addressing is like having bees in your head.

The address mapping problem would be ameliorated if chips kept up with CPUs in address bits† (how many 32 bit peripherals have you seen?). The logically separate pools of memory problem would be ameliorated if, in addition, chips had better built in caches, that is, the chips were able to plug into a common bus, and provide the appearance of shared memory, caching the bus transfers on-chip.

Another class of problem arises when chips designed for multiplexed address/data busses are used in systems with non-multiplexed busses. In this case, the software must first write, as data, the address of the on-chip register which it subsequently reads or writes. This makes attempts to map the registers via a C structure declaration impossible. Additionally, it requires serializing chip accesses to avoid destructive interference. The Zilog Z8530 SCC is an example here too.

Of course, we have been assuming here that all I/O is memory mapped, à la PDP-11. CPUs which require special I/O instructions have all these problems and more.

Address Decoding

Hardware designers often skimp on the logic which decodes addresses to select chips. This is understandable, since to meet the software ideal of a one byte register occupying exactly one byte anywhere in the address space requires, on a 32 bit bus, a 32 bit comparator and 32 switches. Worse yet, it is often the case that accessing a non-existent address does not return an error. This lack of exact addressing, like the lack of strong type checking, makes software difficult to debug. An extreme case: On the Sun-1 CPU board, a stray write access has a 1 in 16 chance of disabling parity checking!

---

† Admittedly, there are quantum leaps in packaging costs associated with increasing pin counts just past 28 or 40; this tends to contribute to the size-of-address-space problem.

Timing

Many chips specify that they can only be accessed every so often, such as the Zilog Z8530 SCC, which has a "write recovery time" of 1.6 micro-seconds. Enforcing this recovery time in hardware requires several extra chips between the CPU and the SCC, so the hardware designers let the software enforce it. Implementing these very short delays is hard to get right, especially from a driver written in C. Also, the code which implements the delay is dependent not only on processor type, but also on processor clock rate. There are also instances in which is it not clear what the delays need to be; these must be determined empirically, a difficult and error prone dark art. The typical result is that the delays are much longer than what is actually needed, which affects throughput and efficiency. Chips should enforce their delays internally using a bus handshake. An example of an unsuccessful attempt at this is the National MM58167 chip.

Symmetry

Another problem that arises when employing some chips is there lack of symmetry, in which an identical operation must be handled in a unique way, one for each resource. The reset operation for the channels of the Zilog Z8530 SCC illustrates this, since, although the channels have the same functionality, the bit pattern used to reset channel A is not the same for channel B, requiring the software to know on which channel it is performing a reset operation.

Whither the Gurus?

A system-level problem can exist with complex chips. Hardware designers may not understand the system implications of a chip due to the chip's software complexity; and hardware implementors may not be able to adequately test the chip's incorporation in their design because of the large amount of software required. The software developers may understand how to program the chip without knowing what operating modes are appropriate because they don't understand the hardware timing, addressing, and interrupts. The end result is that no one person adequately understands the chip's behavior in the system.

UNIX Uber Alles

Some chips are geared to a particular hardware architecture, handling such specifics as byte ordering. But UNIX is a particularly interesting software architecture that can also take advantage of specific chip features.

Because UNIX is written almost exclusively in C, the chip should support a natural C language interface by allowing C structures to map directly onto control and status registers. This requires that the registers related to a resource are explicit and clustered together; that the registers are accessible by de-referencing a C pointer; and that the registers will read back what was written to them, to allow bit field operations. And since UNIX uses a memory mapped model for device accesses, the chip should be byte-, word-, and long-word addressable and accessible. Having the mapped memory behave like shared memory, with only one writer, greatly facilitates kernel-chip interactions by affording a simple, powerful protocol.

Since UNIX is a multiprocessing system, the chip should be quickly and easily context switchable. This implies that there be no hidden state, and that there be a fast and easy way to save and restore the chip's context. For performance, it may be worthwhile to provide an indication of state change (much like a modified bit for a virtual page). Obviously, the less data saved or restored, the faster the context switch will be. For example, we may save almost 300 bytes for the Motorola 68881 floating point chip on a context switch, 96 bytes for registers and about 200 bytes for internal state and software status. One way to reduce the average context switch time is to have multiple contexts on chip, and allow UNIX to allocate them to processes.

UNIX interrupt processing overhead can be reduced by the use of unique interrupt vectors, which decrease the need to poll in order to determine the reason for the interrupt. If there is a need to poll, the chip should make determining the reason quick and convenient.

The interrupt lockout mechanism of the UNIX kernel, combined with us lazy programmers, leads to very high interrupt latency. Allowing longer interrupt latency makes driver and kernel writing easier. Therefore, the chip should provide sufficient buffering to tolerate an occasional long delay in interrupt servicing; three characters of buffering is not enough for a UART; at 9600 baud, that is only about 3 milliseconds. Although this is much longer than just the usual interrupt overhead, it does not allow much time for the interrupt routines themselves.

Since UNIX automatically configures itself, a software probe routine is necessary. The probe routine should be easy to write and robust, implying that the device should have some readily testable characteristic (which should be distinguishable from regular memory and most other devices). In addition, a desirable feature on powerup is for the chip not to speak until spoken to, that is, to present no interrupts until specifically enabled. This would allow UNIX to determine the interrupt priority and vector location at probe time —— we are currently unable to do this at Sun.

Conclusions

With the tendency to pack more features per microacre, more direct chip-connects are being done to avoid glue chips. Since the glue chips are gone, there is less insulation from the complex chip and its quirks, like timing or address decoding. So what features should these complex, space-saving chips offer in order to provide the best system architecture, which is not necessarily the smallest and densest board? In general, the chips should provide orthogonal features and avoid special cases and oddball features. (Violations of this maxim account for much of the difficulty in programming anything.) The chips should lean toward elegant simplicity, implementing features to complement software, not usurp it; the chips should provide enough address bits to make the likely direct connect safe and convenient, and enforce (small) timing restrictions internally. For UNIX in particular, the chips should support a natural C language interface, using memory mapped accesses; they should be readily identifiable for a probe routine; they should provide fast and convenient context switching, and support distinct interrupt vectors; and they should tolerate occasional interrupt servicing delays.

Acknowledgements

This paper draws heavily on our experience at Sun Microsystems, as well as the experiences of our colleagues in the systems group. We acknowledge their ongoing contributions. Andy Bechtolsheim packs more functionality per board than anyone else we know, always striving for all the chips that fit. If he had not used such chips in his designs, we could not have had such close encounters. Eric Schmidt, director of software, has been instrumental in creating an atmosphere in which real world requirements are met while engaged in interesting and satisfying professional engineering endeavors. We also wish to thank Alison Shanks for late night proofreading of an early draft of this paper.

The Hideous Name

Rob Pike
P.J. Weinberger

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The principles of good naming in computing have been known for decades, and partly because it follows them, the UNIX† operating system is unusually easy to use and to extend. The invention of new facilities can be guided by these principles, so that merely providing a file system name for a service, for example the Eighth Edition file `/dev/stdin`, determines many of its properties. The development of networking has not introduced any change to the majority of UNIX system utilities because objects such as files on remote machines can be given syntactically familiar names within the local machine's name space. UNIX systems are now being connected to outside networks that have complicated naming schemes. As a result, UNIX software must be changed to cope with irregularly constructed names. The practitioners of internetworking would profit by understanding the benefits of simple, uniform syntax.

research!ucbvax!@cmu-cs-pt.arpa:@CMU-ITC-LINUS:dave%CMU-ITC-LINUS@CMU-CS-PT
- Carnegie-Mellon University mailer

I cannot tell what the dickens his name is.
- Shakespeare, *Merry Wives of Windsor*, II. ii. 20.

Any object relevant to computation — file, process, user, computer, network or whatever — needs a name. The name determines the access: it is by interpreting the name that a program (say) is given access to the object. The manner in which names are constructed affects not only how objects are named but also how they are used. A UNIX file name, for example, is a path from one node to another in a tree of names (the name space). The representation of the name is a simple ASCII string, and only by interpreting the string as node names separated by slashes is the file actually identified. The structure of the name space (a directory tree) is reflected in the style of the name (a path through the tree). Were the file system arranged differently, say as a flat array, the form and interpretation of file names would also be different; for example, UNIX processes are named by small integers.

The form of the name space is the subject of this essay. We will use file names in various operating systems to illustrate criteria for distinguishing good names from bad ones. These criteria are then applied to network mail names, including 'standard' ARPA Internet mail names. The criteria are not new, but seem to have been forgotten, so it is worth attempting to re-establish them.

Name spaces have some general properties. First, names within the space may be absolute or relative. Absolute names specify an object by position with respect to a single fixed point, such as the UNIX file system 'root' (named /); relative names, with respect to a local point (the 'current directory,' named .). Also, a name space typically has operators to change the space,

† UNIX is a Trademark of Bell Laboratories.

such as the UNIX `creat`, `unlink`, `link`, `mount` and `chroot` system calls. Finally, a name space has syntax — how a name is constructed — and semantics — the nature of the object a name identifies.

A UNIX file name, for example, is a sequence of slash-separated strings that identifies a byte stream. External conventions may provide further semantics: the file system contains objects that are not ordinary files. Simply by having ordinary file names, though, these objects have ordinary file properties such as protection. Examples are device files (`/dev/tty`), and in the Eighth Edition, processes (`/proc/01234`), faces (`/n/face/research/pjw`) connections to other processes (`/dev/pt/pt04`), and even synonyms for other files (`/dev/stdin`). Because they have regular names, existing tools can treat them as files when convenient, so standard software such as `open`(2), `read`(2), `cat`(1) and `ls`(1) immediately provide services for these objects that would otherwise require special handling.

Some of the Eighth Edition examples above have different names in other UNIX systems. `/dev/stdin` is often represented by the single character `-`, as an argument to commands, but this convention is capriciously followed; because it must be provided explicitly by each program, it is only available in some programs (compare the shell metacharacters for matching file names). Processes are represented by an integer process identifier, which is only meaningful to a few process-specific system calls. These calls implement their own protection mechanism, although the protection provided by the file system suits perfectly. Virtual terminals implemented using the multiplexed files of the Seventh Edition have no external name, so it is impossible to open one for IO (consider `write`(1)). On the other hand, a name in the file system is available, without prearrangement or special protocol, to any program. Also, clean syntax allows transparent extensions of semantics.

When machines are connected together, their name spaces may be joined to facilitate sharing files. If the name spaces have the same clean structure, that structure can be extended simply to describe the larger space. The Newcastle Connection names a file on another machine, say ucbvax, as `/../ucbvax/usr/rob/bin/cat-v`; the Eighth Edition notation is `/n/ucbvax/usr/rob/bin/cat-v`. In the former the name space has been extended by making it a subspace of a larger space, in the latter a new name subspace has been grafted on using `mount`(2), but in neither case has the *syntax* of names been changed; any program that understands a file name will understand a network file name without change, and relative names for files (those that don't begin with /) are unchanged. As a spectacular example, we might see on which machines wnj has a login by grepping through `/n/*/etc/passwd`, or even on those machines connected to ucbvax by `/n/ucbvax/n/*/etc/passwd`. These systems might have no global root, so meaning of an absolute name may become ambiguous because of the presence of multiple reference points. In fact, there might be no single point to which all names can be fixed. In practice, though, this ambiguity is unimportant.

Unfortunately, not everyone chooses naming conventions in accord with these (implicit) guidelines. An easy target is VMS, where our canonical file might be called `UCBVAX::SYS$DISK:[rob.bin]cat_v.exe;13`. The VMS file naming scheme provides a distinct syntax for each level in the name: `UCBVAX::` is a machine; `SYS$DISK:` is a disk (actually a macro that expands to a disk name such as `DUA0:`); `[rob.bin]` is a directory; `cat_v` is a file 'base' name; `.exe` is a file 'type'; and `;13` is a version number. This syntax is analogous to programming languages; consider a C expression such as `*struct[index].field->ptr`. If `*` were postfix and `/` the only dereferencing operator, the expression might be written `struct/index/field/ptr/`. Functionally-minded programmers might use the notation `contents(ptr(field(index(struct))))`. (A single character cannot be used in C because it could not distinguish `X[Y]` and `X->Y`, with `X` a structure pointer and `Y` an integer or structure element respectively, but this ambiguity could be eliminated in a different language.) C and VMS use syntax to distinguish the types of the components of a name; the UNIX file system instead deli-

berately hides the distinctions. Aside from the obvious advantages such as simplicity of syntax and the usurping of only a single character, the uniformity also makes the name space easier to manipulate: the `mount` system call aliases a disk and a directory.

VMS has no true name space manipulation operator. Although one could construct one, it would be limited in scope: how could a disk be mounted at the VMS equivalent of `/usr/src` when disks are always before directories in the name? Instead, VMS has macros like `SYS$DISK` to hide the manner in which the space is assembled, and even to provide the concept of a local name by automatically inserting an expanded macro before an unqualified name. The problems with dynamic evaluation of macros are well known; for example, the VMS equivalent of `chdir` sets the default prefix for file names, but the prefix will only be evaluated and so checked for validity when a file name is interpreted, which may be arbitrarily and confusingly long after the prefix was set. Also, these local names are not really local at all, but rather implicitly prefixed by a string that binds them to a root of the name space. This implies that all names are always attached to some root, and therefore if the root changes, the name must also change, invisibly. (In fact, the default prefix macro is handled in a special way, because directories are not constructed by simple concatenation; subdirectory `[bin]` in directory `[rob]` is named `[rob.bin]`.)

Another problem with VMS names is that one cannot do the equivalent of grepping the VMS password files on various machines with `*::SYS$SYSTEM:SYSUAF.LIS`; the `*` operator doesn't apply to that portion of a name. This is an example of the general problem that whenever the name syntax is changed all programs that interpret names must be modified. More subtly, although if ucbvax were a gateway we could access files on a distant machine as `UCBVAX::KREMVAX::file`, it is only because the semantics of `::` explicitly permit such access. The `::` operator is implemented by passing the string after it to the remote machine, but first checking it for syntactic validity, so the syntax checker must have special code to admit multiple `::`'s.

The Cedar file system has a uniform naming syntax, just like UNIX file names, except that the version number of a file is separated from the file name by an exclamation mark `!`. The implementers apparently thought that version numbers are fundamentally different components of file names and therefore deserved different syntax. But the change in syntax requires new rules to define the meaning of file names. A good test of naming schemes is whether arbitrary names constructed by the syntactic rules make intuitive sense without new semantic rules being created for their interpretation. In the Cedar file system, `/usr/rob/bin/cat-v!3` is clearly version 3 of `cat-v`, but what is `/usr/rob!3/bin/cat-v`?

Even our friends sometimes make mistakes. The IBIS remote file system on 4.2BSD names a remote file as `ucbvax:file`. Many programs don't understand this syntax; we are forced to change the shell if we want to make `*:file` behave as we expect, because the shell expects a slash to separate name components. Worse, by changing the syntax, the implicit semantics of the original naming scheme is lost. In the Eighth Edition name `/n/ucbvax/file` it is obvious what `file` refers to: a file in the root directory of ucbvax. But what is it in `ucbvax:file`? It *might* be a file in the root, but it isn't. It is a file in the *home* directory on the *destination* machine (ucbvax) of the user invoking the name on the source machine (unless it begins with `/`); its meaning depends on who is asking. The extra semantics of `:` complicate attempts to patch the syntactic problems. We might try creating a symbolic link `/n/ucbvax` that evaluates to `ucbvax:`, but `/n/ucbvax/file` still points to a file in someone's home directory, and `/n/ucbvax/usr/wnj/file` refers to `/usr/wnj/usr/wnj/file`. If the link evaluates to `ucbvax:/`, things work as expected, but (we note without regret) the slash-less form of IBIS naming is made unavailable.

Part of the problem in the IBIS file system is that it is implemented outside the name space. By using a variant of `mount`(2), the Eighth Edition file system guarantees that the syntax and semantics of names are free of surprises. For example, it is clear what `/n/ucbvax/n/kremvax/file` refers to, but what about `ucbvax:kremvax:file`? Where does `kremvax:file` get interpreted?

There are other ways to interpret file names like `ucbvax:file`. When using uucp to copy a local file to a remote machine, the name `ucbvax!file` refers to the file on ucbvax whose name is `file` prefixed by the current directory on the *source* machine. The prize goes to DECNET, however: `ucbvax::file` refers to `file` in the home directory of the 'default network user' on the destination machine, and `ucbvax"wnj password"::file` refers to `file` in wnj's home directory (yes, the password is in clear text).

In summary, there are some guidelines for constructing naming conventions, particularly for objects in a network. There should be both relative names and absolute names. Relative names are actually more important because, among other reasons, the root of the name space may be unknown or non-unique. The syntax should be clean and uniform; every new syntactic rule requires at least one, and usually many, semantic rules to resolve peculiarities introduced by the new syntax. If the name space is a tree or any other kind of graph, a single character should be used to separate nodes in a name. If these guidelines are followed, names of objects in a network of machines will be easy to construct and interpret; difficult problems of networking will be completely hidden to the users and programs accessing objects in the network. If they are ignored, both users and programs must be aware of and understand the details of naming locally, globally and everywhere in between.

So far our examples have been of file names in file systems, but the same principles of good names apply to the names given to recipients of electronic mail and other inhabitants of computer networks. Since names constructed by our guidelines can name an arbitrary file, with arbitrary contents, the rules of simple naming are sufficient to name anything at all. It is interesting to see how well the guidelines are applied in practice.

Mail sent by UUCP is addressed by names parsed from left to right: `place-1!place-2!...!final-place!user` is a relative name that uses the one special character `!` to separate components. The interpretation is simple, uniform, and not necessarily tied to routing.

The state of network naming outside the UUCP community is somewhat more complex. To deal with the problems that developed as more networks began attaching to the ARPANET, the concept of a 'domain,' generalizing 'host,' was introduced. The idea is that rather than a route through a set of hosts, a name in the network contains a sequence of subdomains that form an unambiguous path through an organizational hierarchy, prefixed by a name understood in the innermost domain. New concepts need new syntax, someone thought. The syntax for domains is a string of identifiers separated by dots, as in wnj@ucbvax.world.flat.BIG. In the so-called standard (as of August 13, 1982; it changes underfoot) the relevant part of an address is `local-part@domain` where both the local part and the domain are dot-separated lists of words. The local part can be anything that the first domain to the right of `@` understands, while the domain string has a clean and standard interpretation. How well do these names follow our rules?

First, according to RFC 882 ("Domain Names - Concepts and Facilities"), the domains are all absolute. The dot signifying the root of the hierarchy is implicit at the right of the list of names, which makes it impossible to connect disjoint name spaces with this scheme since all interpreters of names must know all names at the top level of the hierarchy. Also, for backwards compatibility, RFC 822 ("Standard for the Format of ARPA Internet Text Messages") allows all but the leftmost of the domain names to be elided, since "specification of a fully qualified address can become inconvenient."

Second, there are at least two special characters, '`@`' and '`.`', although the `@` serves no purpose. The only legal way of interpreting RFC 822 addresses is to deal with the rightmost domain name, and pass the rest on to that domain. This stops at the leaves of the domain tree, or the `@`, which is the same thing.

Finally, the standard requires 'registered' domain names: all the names that occur as domains in valid names must be registered before use, thus arrogating to ARPA Internet administration the authority to legislate naming. As a result, the whole thing is encompassed in a closed society and there are very few registered names. So much for internetworking.

Let us examine a typical Internet name:

IJQ3SRA%UCLAMVS.BITNET%SU-LINDY@SU-CSLI.ARPA

This is the name of user IJQ3SRA on machine UCLAMVS, accessible through BITNET from machine SU-LINDY, which is known to SU-CSLI on the ARPANET. There are obviously at least two domains in this name, BITNET and ARPA, but only one syntactic domain. Instead of using the domain scheme, the gateway service between BITNET and ARPANET was made explicit in the name, requiring the invention of a new syntax character (%) which is translated to @ at the gateway, because ARPANET names can only contain a single @. A more rational standard name would be

IJQ3RSA.uclamvs.su-lindy.su-csli.bitnet.arpa

but that wouldn't work, even were the two Stanford sites separated by @ instead of a dot, because BITNET is not a registered name. If it were, one would omit .su-lindy.su-csli. Also note that the local part of the name above contains source routing — a path specified in the address by the originator of the message — despite the intention of domains to avoid routing. RFC 822 has an alternate syntax that allows the specification of routes by introducing the three new characters ⟨, : and ⟩, and rendering the % unneeded. (We nearly leave out mention of the use of lower and upper case, except to observe that the local part is case sensitive and the domain part is case insensitive, except that all 1024 case variants of the reserved local part 'postmaster' are legal and equivalent.)

What is going on? Basically, despite the words in the standard about hierarchy, the domain space is completely flat, so the local parts of the names carry source routing and domain transitions explicitly. To worsen matters, machines that advertise adherence to the standard in fact do not; instead the name translations that occur at gateways (such as converting @ to % and rearranging the components) are at best *ad hoc*. Names like poor IJQ3SRA%UCLAMVS's above are simply not following the rules.

No one pays any attention to the standard as long as the software keeps delivering mail. This means that a mail transmitting program in practice must understand the details of local names outside its domain, to the extent that constructing a network mailer is a research topic in heuristic programming.

New syntactic features will be invented to solve new problems, and someday we will have to interpret names even more complicated than

@cmu-cs-pt.arpa:@CMU-ITC-LINUS:dave%CMU-ITC-LINUS@CMU-CS-PT

Doug McIlroy has observed that

> ... bad notations can stifle progress. Roman numerals hobbled mathematics for a millennium but were propagated by custom and by natural deference to authority. Today we no longer meekly accept individual authority. Instead, we have "standards," impersonal imprimaturs on convention. Some standards are sound and indispensable; some simply celebrate bureaucratic littleness of mind. A harvest of gimmicks to save appearances within the standard has grown up, then gimmicks to save the appearances within the appearances. You know how each one got there: an overnight hack to paste another tumor onto a wild cancerous growth. The concern was with method, regardless of results. The result is extravagantly worse than Roman numerals: you can't read the notation right to left or left to right. As an amalgam of languages, it can't be deciphered by a native speaker of any one of them, much as if we were to switch at random places in a number between Roman and Arabic signs and between big-endian and little-endian order. But now that it all "works" — at least for the strong of stomach — the tumors themselves are being standardized.

The UNIX community has a clean, attractive naming scheme. The ARPA Internet community doesn't. Nonetheless, the ARPANET is influencing names within the UNIX community rather than the other way around. We are not proposing that ARPA adopt UNIX file system

naming, we merely ask they they adopt and enforce a simple, sensible scheme. There is a precedent for world-wide names and addresses: through the middle of the 19th century, it became more and more difficult to handle international mail, until the International Postal Convention of 1875 established the standards that still form the foundation of the modern postal service.

It is hard to imagine organizations less likely to cooperate than hostile governments, yet a letter addressed in a form and alphabet locally understood can be sent anywhere in the world. Ignorance of the standard Iranian form of an address doesn't cut Iranian citizens off American mailing lists, because the USA and Iran (and every other pair of countries) have rules about how mail between the countries is handled. Moreover, users of the mail system don't know the rules, only the post offices do.

The problem for electronic mail is smaller, simpler and surely solvable. RFC 822, RFC 882 and their implementation, however, are not the solution. They represent the unilateral imposition of a deficient standard that is polluting those communities that have comprehensible naming schemes.

> I fled, and cry'd out "Death";
> Hell trembled at the hideous name, and sigh'd
> From all her caves, and back resounded, "Death."
>
> - Milton, Paradise Lost

Who Answers Your Telephone When You're in the Information Age?

Brian E. Redman

Bell Communications Research
Morristown, New Jersey 07960

ABSTRACT

In an effort to provide better telephone coverage, a new type of telephone answering machine has been designed. The prototype consists mainly of an off-the-shelf product which has been programmed to answer the subscriber's telephone, accept Touch-Tone * inputs and report them to the subscriber via electronic mail. Work is in progress to incorporate the answering machine functions into a fully functional telephone switch.

1. Introduction

Even in this day and age the telephone is a valuable instrument. The ability to communicate with another party in real time over vast distances is essential to our modern life styles. Although there are times when an unanswered telephone is desirable, typically so you can say "But I called and there was no answer" to the person you didn't really want to speak with, for the most part when you call someone you hope they will answer. Answering your telephone is a courtesy to the caller. It may also suit your purposes to be contacted.

How do you answer your telephone when you're away from it? There are several methods which all involve having something or someone answer it for you. The most common method is to use a conventional answering machine. They are inexpensive and do the job. But even the most elaborate devices available today have several drawbacks.

Many callers do not like to speak to an answering machine.

Commercial answering machines will take a message, but will not contact you when they have done so.

They are incapable of interacting with the caller.

The only current alternative to such an answering machine is an answering person. This unfortunately is a very high cost solution. At one large corporation engaged in telephone research and development the service is provided only to staff members above a particular level. At that institution, while members of technical staff attend meetings, conferences, lunch, or are otherwise out of the office, their telephones go unattended. At another similar institution, although secretarial pickup is provided, the quality of service is dictated by corporate rank. Bemused by this situation, and with no apparent prospects for advancement to a level at which personalized answering is provided, and unsatisfied with the capabilities of available answering machines, the motivation to create the systems described here was great.

*Other brands of Dual Tone Multi-Frequency (DTMF) input are also accepted.

2. The Initial Prototype

2.1 Operation

A simple device was constructed and used (see the Appendix) which works as follows: When the device detects a ringing line, it goes offhook, recites a message ("Please enter your telephone number with Touch-Tone and end with pound."), collects Touch-Tone input, and says either "Thank you. Someone will call you back." or "No number entered." depending on whether it received input. The Touch-Tone input and the line number are then sent to a shell program on a UNIX† system which sends the message as mail to the appropriate subscriber.

The advantages of this combination of telephone-related hardware and a standard UNIX system are:

> The subscriber may get immediate notification of messages if they are logged in to a machine where they receive mail.
>
> A dated record of telephone calls is kept.
>
> Characteristics are easily modified (e.g., number of rings before interception).
>
> Callers who won't "talk to" machines don't seem to mind sending tones to them.

2.2 Deficiencies

The prototype hardware can be improved upon. Since the speech synthesizer uses a stored vocabulary, the spoken message is not flexible enough. For example it is not possible for us to identify the subscriber by name. The device is not interrupted when the caller starts sending tones; it would be desirable for the caller to be able to cut a spoken message short. The ring detector is sometimes fooled by switchhook activity.

The major problem we encountered with the concept is that a telephone number does not always identify the caller. Occasionally we would return a call and experience the following:

> Hello, Large Corporation Employing Thousands of People, may I help you?
>
> Uh, this is Brian Redman from Bellcore, someone phoned me at 2:13 P.M. last Monday and left this number for me to call back.
>
> Who was that sir?
>
> I don't have a name, only this number.
>
> I'm sorry sir, we employ thousands of people here, I need a name to help you.
>
> Thank you, I guess they'll phone again.

This problem is remedied as callers become familiar with the system and leave further identification such as an extension or coded ID. An on-line telephone directory, and a database that indicates what town a phone number is in, have proven useful for identifying callers. See also the discussion of ongoing future work.

2.3 Other Uses

Although the typical use of the this device is to leave telephone numbers, an arbitrary message can be encoded with the Touch-Tone pad. The convention is that two numbers represent a letter; the first identifies the three letter group, the second pinpoints the letter in that group. Other schemes are possible which enable a caller to transmit any ASCII character [1,2]. Experience has indicated that a simple convention is good enough.

---

† UNIX is a Trademark of Bell Laboratories.

This system has also been used to circumvent certain deficiencies in the local voice communications system. For example, call forwarding can only be set up at the extension which is to be forwarded. It is often be convenient to be able to establish call forwarding from a remote station. For instance, if one stays longer in a colleague's office than anticipated and decides to have calls forwarded there, it is natural to want to do this without returning to one's office to set it up. This machine is programmed so that a special code can be input in response to the standard message, to enable forwarding. This code is followed by the number to forward to. When the subscriber hangs up, the machine goes offhook and dials the appropriate call forwarding sequence to perform this operation. The system is used in this example as a subscriber simulator. Other applications which make use of this functional generality follow naturally.

2.4 Usage Data

1700 calls were logged over a seven month period. 52% of the callers left no number (this is represented to the subscriber as "Unknown caller"). This is attributable to human behavior and circumstance.

Some callers do not wish to leave messages. This seems to be independent of the message handling system. An informal survey of secretaries indicates that in 35% of the calls they answer, the caller leaves no name or message. Interviews with callers in a study of their reactions to answering machines [3] reveals that one third of the callers leave no message. However, a study which used actual records of calls to answering machines [4] found that callers failed to leave a message 60% of the time.

A caller's first encounter with a synthesized recording that does not confirm that they have reached the correct number may cause them to hang up thinking they have dialed incorrectly. 8% of the calls were paired; an unknown caller, immediately followed by a caller who left a number, indicating this reaction.

Many callers seem to be so taken by this system that they call several times just to hear it and have their friends listen. Once while testing the system, its output was directed to a loudspeaker. The following transaction was overheard:

⟨ring ring⟩

Please enter your phone number with Touch-Tone and end with pound.

Huh? What the hell is this? ⟨click⟩

⟨ring ring⟩

Please enter your phone number with Touch-Tone and end with pound.

Hey Mildred, come here! You've got to hear this. ⟨click⟩

⟨ring ring⟩

Please enter your phone number with Touch-Tone and end with pound.

It sounds like a computer! Yo! Mary, Steve! Check out this weird. . . ⟨click⟩

⟨ring ring⟩

Please enter your phone number with Touch-Tone and end with pound.

I think it's adorable.
Sounds pretty strange.
Is that really a computer?
What do we do now?
Whose voice is that?
Whose number is this?
What's it mean end with a pound?
What a strange telephone. ⟨click⟩

⟨No number was entered.⟩

This behavior accounts for 4% of the calls.

We observed that 3% of the calls represent callers who made an error in entering their phone number and immediately called back and entered it again.

A small percentage of callers left false numbers intentionally, presumably in attempts at humor. These crank calls would direct the subscriber to dial the White House, various "phone-it" services, or some generic number.

Some callers do not have Touch-Tone service or are confused by the machine. These people would contribute to a slightly larger number of abandoned calls than for a conventional answering machine. On the other hand, the ability to leave a message mechanically would be likely to encourage a higher percentage of messages. These factors seem to cancel each other. Whether the abandoned call ratio is higher or lower than that encountered with other answering devices depends on which study you examine. Further investigation of callers' interactions with various answering mechanisms would be enlightening.

2.5 Comments

Many comments have been received from callers concerning the "unorthodox answering machine". On the negative side, some callers have failed to understand the instructions because they didn't know what "Touch-Tone" or "pound" meant. In one case a caller was defeated by their local PBX which intercepted their tones. At least one caller was repeatedly frustrated by the fact that he didn't have a Touch-Tone phone. He also didn't like the "tone" of the synthesized voice.

There have been far more positive comments than negative. They include the appreciation of not having to "talk to a machine". Many people were fascinated by the system and enjoyed it merely because it was different. One computer equipment salesman said that his batteries drain down over the course of the day and his encounter with the machine typifies the sort of thing that charges him up and keeps him going. Most callers just think it's cute.

3. Recent Developments

The limitation of a single and uninterruptable stored speech source has been overcome by using a more flexible speaking device called DECtalk. This allows the computer to recite more lengthy messages that will be useful for the novice, but that can be interrupted by the experienced caller.

The functioning of the system is now completely controlled by a UNIX program. A *greeting file* is spoken when the machine answers the phone. Caller's actions are confirmed by repeating the digits that were presented. I.e., "*I will send Brian mail to call you back at 213, 641, 6300.*" If a file is found whose name matches the caller's input, that file is spoken. Thus personal messages can be delivered. If the first character of a line is a ' ! ', the remainder of the line is considered to be a UNIX command whose output is also spoken. A typical file might look like:

```
! echo "jim called" ¦ mail ber        # note there is hopefully no output
! echo "'tod' Mr. Phelps."
Your mission, should you decide to accept, is to remove Mr. Big from
control of USENET without alarming the innocent hackers who don't
realize what's going on.  This message will self-destruct immediately.
! rm 801219143              # his social security number, the name of this file
```

The system also allows callers to be transferred to a secretary or a voice recording device. Additionally, a help interaction can be requested.

In order to expand the system's capabilities to provide the types of features described below a telephone switching system was acquired. Currently work is directed towards implementing a feature-rich system where the behavior of a telephone can be tailored by the user. Naturally a bitmapped graphic interface and mouse come to mind.

4. Future Work

4.1 Short Term

Some new and different services are planned. These include a voice announcement feature like that used in the Computing Science Research Center at Bell Labs [5]. It will be enhanced so that any nearby phone may be used to receive a call. It's expected that the subscriber would dial into the system and present identification to have an inbound call switched to them. Speakers and phones for this use will be placed at common meeting points in the corridors and other strategic locations.

The system will also be made more powerful in an aim to help the caller reach the subscriber. For instance the subscriber may wish that certain callers activate a standard "beeper" or that a series of numbers be called in attempt to contact the subscriber. The following scenario is imaginable:

⟨ring ring⟩

*Hello, this is Gordon Woods' answering service. Gordon is not available now. Please leave your number by using the Touch-Tone pad and he will call you back. Press *# if you would like more detailed instructions.*

⟨caller types special code⟩

Ah, Mr. Redman. Gordon has left instructions with me to contact him immediately if you called. Please hold on while I attempt to reach him.

⟨The BERphone makes the necessary phone calls, pages, etc.⟩
⟨it calls Mr. Hodgdon's office⟩

⟨ring ring⟩

Dave Hodgdon.

Hello Mr. Hodgdon, this is Gordon Woods' answering service. I have Mr. Redman on the line and Mr. Woods asked me to try to reach him there. Is Mr. Woods available?

⟨Hodgdon indicates affirmative by pressing "1"⟩

⟨Woods types in his ID and the call is switched to him⟩

Any aspect of this scenario should be programmable at any time by the subscriber from any location. Many parameters can be easily set using only a telephone.

The system could also provide a semi-automatic call-back service. The subscriber would direct it to dial the numbers which it collected.

Experimentation with the use of the system to access general services from the host computer such as mail and news are envisioned. Perhaps a caller would like to play a game while on hold? Or choose the type of music, or listen to a weather report ...

The possibilities are enormous, approaching the simulation of a personal secretary standing by the subscriber's phone 24 hours a day.

4.2 Commercial Possibilities

The system is implemented as an add-on to the existing telephone system by bridging the subscribers' lines. Similar systems (i.e., ones that utilize electronic messages) are being marketed which have recently come to our attention, but they require human attendants and are not capable of sending messages to the subscribers via electronic mail. The system described here would also be very attractive if it were incorporated into a Centrex or PBX system. In this event much of the hardware cost could be saved while the services could be expanded.

Another possibility is that the system could be scaled down. It could still provide many of the features suggested, yet be contained in the telephone instrument itself. Additionally, the telephone could be designed to automatically provide the the caller's own phone number in response to a query from answering equipment at the called number.* Preceding the voice answering message with a short, standard tone sequence would greatly simplify this feature. This arrangement would put an end to the "no message" phenomenon described above. It also provides a mechanism for calling number identification which does not require CCIS in the local central office. When local CCIS does become available, the same protocol could be used to transmit the calling number information from the office to the subscriber before the call is cut through.

*Readers fortunate enough to be familiar with the Model ASR-33 teletypewriter[6] will recognize this as a great idea, but not a new one. The feature known as the WRU (who are you) answer-back caused the machine to transmit the identity of the local station in response to the receipt of an ASCII ENQ (enquiry) character from the remote end. The transmission (which was encoded by breaking tabs off of a cylindrical drum) could also be initiated locally by pressing the HEREIS key.

The widespread deployment of automatic answering systems would generate increased revenues for providers of telephone service and equipment as new products could be marketed, the percentage of completed calls would rise, and callbacks would be generated.

5. What's this have to do with UNIX?

Aside from the fact that it is developed on and is running under UNIX, perhaps some similarities can be drawn. The idea behind this system is to allow users to exploit the capabilites of the telephone network. The system provides a featureful interface but does not limit the capabilities of the underlying resource. The interface can be individually customized. Complex capabilities are assembled from specialized units enhancing implementation of new ideas and modification of existing ones. Although the default behavior will look like a standard telephone interface, it's clear that this system will attract the more sophisticated and enthusiastic telephone users. It's envisioned that users will continue to feed the system with improvements.

6. Conclusions

Experience with this system shows that it provides a useful service and that the basic principles can be enhanced to provide exceptionally good telephone service at modest cost. Enhancements should allow subscribers to make better use of the telephone network and allow them to independently control how their telephone is answered and how the call should progress. It is also felt that such services will some day be commonplace and therefore a market for new types of automated answering devices, switches and the like has been identified. Communication is the keystone of the Information Age. Answered calls strengthen communication.

Acknowledgements

This work was performed with Dave Hodgdon and Gordon Woods who work at a large corporation engaged in telephone research and development. Were it not for certain practical considerations they would appear as co-authors. We wish to thank Pat Parseghian for being our test subscriber and putting up with the system through its unstable periods. And although people who call us can expect odd things to happen, we particularly thank Pat's callers who were caught off guard.

References

[1] Lauren Weinstein, A Touch-Tone Input and Voice Output System for UNIX. Vortex Press, February, 1976.

[2] Martin Minow, Touch-Tone Access to RSTS/E. Digital Equipment Corporation internal memorandum.

[3] R. J. Bowers and S. J. Johnson, A study of Callers' Reactions to Answering and Recording Devices, 38933-11, Bell Labs Memorandum for File, November, 1978.

[4] A. J. Kames, An In-House Trial of Telephone Answer and Record Services, 38736-41 and 39115, Bell Labs Memorandum for File, December, 1975.

[5] Joe Condon and Ken Thompson, CallFor, the voice announcement part of TPC. Bell Labs internal phenomena.

[6] Teletype Model ASR 33 User's Manual. Teletype Corporation, Bulletin 310B, Vol 1. Technical Manual, 33 Teletypewriter Sets. October, 1971.

Appendix I - Apparatus

The system that we put together (figure 1) utilizes a SLC II from Digital Pathways, a custom ring detection circuit which is controlled by an LSI 11, and a UNIX host.

The SLC II incorporates a speech synthesizer (stored vocabulary), a DTMF (Touch-Tone) decoder, an automatic calling unit, and serial ports. There are several features such as parallel ports, clocks, timers, and others that were not used in our system. The unit is programmable via assembly language, BASIC, or its own command interpreter. We only used the latter.

The ring detection circuit was built so the SLC II system could accommodate up to sixteen subscriber lines. Subscribers lines are bridged to it. The circuit will connect any of these lines to an output line which is the interface to the SLC II. Ringing lines are indicated via a 16 bit parallel interface.

The LSI 11 is connected to the ring detection circuit by means of a DRV11-C. It is programmed such that a ringing line triggers an interrupt routine which transmits the line number in ASCII to the SLC II and causes the line to be switched through.

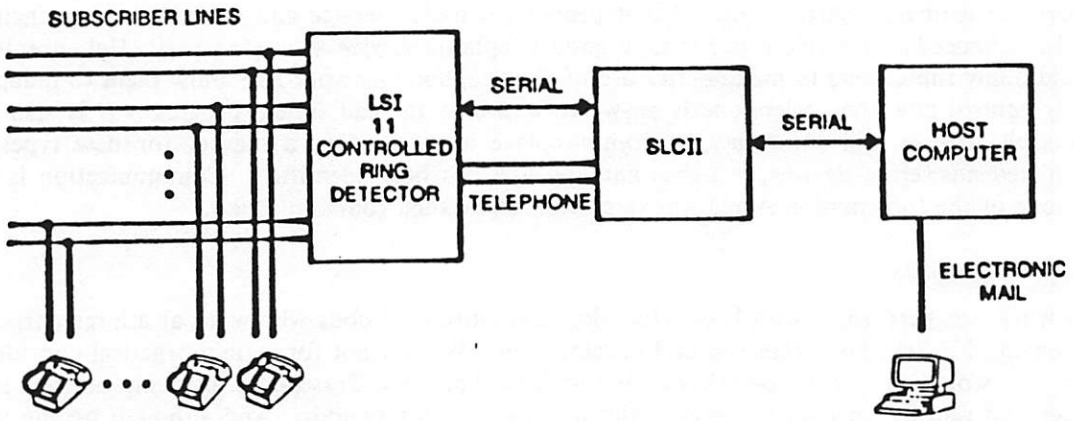


FIGURE 1

A Portable Intermediate Code Optimizer for C

Thomas J. Kelly
Allen McIntosh

Human Computing Resources Corporation
10 St. Mary Street
Toronto, Ontario, Canada
M4Y 1P9
(416) 922-1937

ABSTRACT

The C compiler for a UNIX* system is often based on the Portable C Compiler (PCC). One of the strengths of PCC is its portability; one of its weaknesses is the quality of the code that it generates. An assembly language level optimizer is often used to improve code quality, but there is a class of more global optimizations that are better done earlier in the compilation process. These optimizations include elimination of common subexpressions, constant folding, allocation of variables to registers, and loop invariant removal. This paper describes an optimizer capable of making these and other transformations. The optimizer transforms the intermediate code output by the first pass of PCC, and so is portable to any machine with a PCC-based compiler. Since the UNIX FORTRAN 77 compiler uses the same intermediate code, the optimizer can be used on FORTRAN 77 programs as well. Our paper focuses on the overall design and implementation of our optimizer, with particular attention to correctness issues. Results obtained compare well with other currently available technology, suggesting that the existing investment in PCC-based compilers need not be abandoned.

1. Introduction

This paper describes PCO, a portable C optimizer. (Like most "optimizers", PCO does not attempt to produce an optimal program; it transforms an input C program into an equivalent but more efficient C program. We bow to tradition in calling it an optimizer). PCO derives its portability from the fact that it operates on a machine independent representation of the program. The transformations it makes are also machine independent, although the decision as to whether a given transformation is beneficial is based on machine specific parameters.

Although each of the transformations made by PCO could be made to the original source program by the programmer, there are several advantages to using the optimizer. The changes can adversely affect readability or portability. The analysis of which transformations are safe is tedious, and must be done again if the program is changed. It is much easier and safer to have this analysis done mechanically. PCO uses information about the target environment to determine whether a given transformation is effective, relieving the programmer of the burden of doing it for each machine to which the program is ported.

PCO is useful for the same reasons a compiler is useful: to amplify human effort. The programmer is freed to concentrate on the more important and difficult problems of algorithm selection and program design; code tuning can be done by the machine. PCO's existence makes it practical to write programs that depend on an optimizing compiler to achieve performance goals, and

---

* UNIX is a trademark of AT&T Bell Laboratories.

the existence of those programs in turn makes use of an optimizing compiler attractive.

PCO was originally developed for a target machine with a RISC-like architecture,[1] and the machine specific decisions are currently tuned for that machine. We have also ported a version to the VAX-11*, and are in the process of tuning it for that architecture.

2. *Project Overview*

The goal of the PCO project was to construct an optimizing C compiler for production use with UNIX System V. Correct implementation of C in this environment was of primary importance, including programs that use signals, `asm()` statements and other difficult-to-optimize facilities.

Since the project was a development project, we decided to use standard optimization techniques and chose the design presented in chapters 12 through 14 of Aho and Ullman[2], as our foundation. Changes and additions to this basic design were required. C has several unusual operators (e.g. `?:`), and C programs make particularly heavy use of pointers. Programmers can request that variables be stored in machine registers. In the UNIX environment, C is used to write device drivers, signals provide asynchronous events, and other interesting facilities are provided. In this paper we focus on problems that arise in optimizing C programs in a UNIX environment and the modifications we made to the basic design to solve these problems.

Our major requirement was correctness: C programs (including those using certain common non-portable techniques) must not have their behaviour changed by PCO. Our goals were compact and fast code. In some cases, speed was preferred over compactness, for example, in lifting invariants from loops. Since the compiler is intended for production use, speed of the optimizer was also of concern.

We implemented PCO as an additional pass of PCC. A PCC-derived compiler exists on many machines, and is often the compiler of choice when doing a UNIX port. Using PCC allowed us to concentrate on optimization issues rather than building a new C compiler from scratch. It also eliminated the risk of accepting a slightly different version of C than the one expected by UNIX programs. To ease porting PCC to new machines, we minimized the changes to PCC where possible.

The use of an assembly language optimizer (such as the Portable Code Improver in UNIX System V) is assumed. We have not included in PCO the program transformations that are normally done at that level, such as eliminating branches to branches. We did have to include some optimizations that were already attempted by PCC pass 1. For example, although pass 1 does some constant folding, PCO can do more by virtue of its additional analysis.

3. *Compiler and Language Issues*

3.1 *The Portable C Compiler*

PCC is structured as a two pass compiler[3, 4] and the passes may be compiled separately. Pass 1 is responsible for syntactic and lexical analysis. It produces a sequence of expression trees, information about register usage and stack frame size, and in certain special cases, assembly code for the target machine. The bulk of code generation is the responsibility of pass 2. When the two passes have been compiled separately, pass 1 writes its output to an intermediate file that is read by pass 2. PCO was designed to read this intermediate file, transform it, and produce another intermediate file in the same format.

---

* VAX is a trademark of Digital Equipment Corporation

A number of changes were necessary to PCC in order to make it compatible with PCO. The changes are localized, and are not difficult to make. Pass 1 originally generated assembly language code for the switch statement, unconditional branches, and labels. All these represent information that PCO must have in order to determine flow of control in a program. In the interest of machine independence we moved the assembly code generation to pass 2, and changed the intermediate file representation. We also moved the generation of assembly code to load parameters into registers from pass 1 to pass 2 so that PCO can promote parameters to registers. Pass 1 generates assembly code only for the function prologue and epilogue. These are not examined by PCO.

Pass 1 originally turned references to parameters and automatic variables into the appropriate register-plus-offset arithmetic involving the frame or argument pointer. This leads to better code since constant subscripts and structure offsets can be folded in, but discards vital information. In this format, there is no distinction between an element of an automatic array, which is not eligible for promotion to a register, and a scalar, which is. Our solution was to delay the transformation to register-plus-offset form until pass 2.

Pass 1 often discards or obscures type information. Type casts may be represented by altering the type of a tree node rather than by a node denoting type conversion. Conversions may be done implicitly, by misrepresenting types of variables. We changed pass 1 to produce information that PCO uses to build to recover these conversions.

3.2 Pitfalls of C

The interface between the C language and the UNIX system contains a number of pitfalls for the unwary compiler writer. The UNIX C library contains functions whose semantics cannot be expressed directly in C. For example, a single call to the `setjmp()` function may appear to return twice with two different function values. The second return is the result of a non-local goto executed subsequent to the first return (via `longjmp()`). Local variables changed between the first and second returns must retain their changed values. Thus, in the following (simplified) code fragment, `j` will be assigned the value 6, not 10, after the second return:

```
i = 5;
if( setjmp() == 0)
   i = 3;
else
   j = i * 2;
```

In addition, `setjmp()` does not work correctly on many implementations. After the second return, machine registers may have the same values that they did at the time of the *first* return. Experienced C programmers know this, but an optimizer might decide that the program would be improved by promoting variables to registers. In any case, these problems are restricted to the function in which `setjmp()` was called. PCO does not optimize any function which calls an anomalous function like `setjmp()`.† Code for the function is written out exactly as it is read in.

The `asm()` pseudo-function allows programmers to insert assembly language instructions into a function. These insertions are often made based on assumptions about how the compiler allocates registers and local variables, and are frequently used to do something that cannot be done easily (or at all) in C. It would require highly machine-specific code for PCO to determine the effect of such code and to include it in its representation of the program. Without making this effort, any optimizer that re-arranges code or the allocation of variables stands a good chance of rendering the program incorrect. For these reasons, PCO does not optimize C functions that

† The `vfork()` function which appears in some UNIX systems is a similar example.

contain assembly code.

In the C language it is impossible to specify that a function has a variable number of parameters. Standard practice in many UNIX programs is to take the address of the first parameter, then increment the resulting pointer in order to access each subsequent parameter. This technique is not portable, but works on many machines. We wanted these programs to continue to work after being optimized. These parameters must not be promoted to registers, so PCO determines if the address of *any* parameter has been computed. If so, it assumes that *all* the parameters might be addressed. This makes them ineligible for some optimizations, including promotion to registers.

Certain C operators (`&&`, `||` and `?:`) imply conditional evaluation of their operands. Although they are represented in the intermediate file as operators, they are in effect flow-of-control. We decided to maintain the operator representation. This decision is discussed in more detail below.

The UNIX `signal` mechanism permits the asynchronous execution of C functions. This means that the value of a global variable (or anything accessible via a global pointer) might change between one statement and the next without any intervening store or function call. References to device registers on machines with memory mapped I/O have the same property, so do references to memory shared with another process. However, C does not specify how asynchronous events interact with the evaluation of operands within a C expression. Hence, asynchronous function calls may be modeled as occurring only between expressions. If an optimizer that generates correct code in the absence of asynchronous events behaves as if a function call occurred between every consecutive pair of expressions, it will also generate correct code in the presence of such events. This is the strategy followed by PCO. For programs that do not use these facilities, this approach unnecessarily restricts the number of possible optimizations. PCO provides an option to specify that asynchronous changes to global variables do not occur.

The proposed ANSI standard for C provides a `volatile` attribute to describe objects that can change asynchronously. This attribute would permit PCO to determine which objects might change; the others could be treated normally. This would eliminate the need for the user option.

4. C Program Representation

One of the key aspects of PCO is the representation of the program. This representation is analysed to determine possible and desirable program transformations. Transformations are performed upon this representation and the output of PCO is generated from it. Since PCO does not attempt inter-procedure analysis, only one function at a time needs to be represented. Since the analysis is done on a per-function basis, the entire function must be represented.

PCO begins its work by reading the intermediate representation of the function. Implicit type conversions in the expression trees (e.g., caused by an assignment) are replaced with explicit conversion operator nodes. The intermediate code is then partitioned into basic blocks, each with one entrance and one exit. A block may exit via a conditional or unconditional branch, a switch branch, or it may "fall through" to the next block.

Each basic block is represented by a directed acyclic graph (DAG). Our basic DAG follows Aho and Ullman[2]. Leaves represent variables or constants, and interior nodes represent operators. Nodes can be labeled with identifiers to indicate that the identifier has the value represented by the node. A DAG is very similar to a collection of trees, but common subexpressions are shared.

The flow graph is used to represent the flow of control in a function. Each node of the graph is a basic block. Directed arcs between nodes show possible control transfers. A flow graph is constructed and checked for reducibility. If the graph is not reducible, further work on the function is abandoned, and the unchanged code is output to the intermediate file. Aho and Ullman[2] state that some of the loop transformation algorithms are not directly applicable to irreducible flow graphs. To maintain our basic design decision to follow Aho and Ullman[2], we decided not to transform those functions. To date, we have run into one irreducible flow graph.

4.1 Loops

After the flow graph has been constructed, PCO analyses it for loops. Our first attempt at loop detection using algorithm 13.1 in AHO ran into difficulty. Loops found by this algorithm may share a common header (consisting of an arbitrary number of basic blocks), and yet each of these loops may contain blocks not found in any other loop. Unfortunately, the register allocation scheme that we adopted requires that the lifetimes of the temporaries that we create to hold loop invariants be either disjoint or nested. This is possible only if the loops for which they were created have the same property.

The following code fragment illustrates another problem:

```
while( ... ) {
   switch(c) {
   case 'A':
      ...
      break;
   case 'B':
      ...
      break;
   }
}
```

The algorithm renders the `while` loop as multiple loops, one for each `break`-terminated sequence of code in the switch statement. An algorithm that removes loop invariants must either create a separate basic block for initialization of loop invariants (called a pre-header) for each loop, or merge the loops and treat them as one.

The first approach results in new loops that are nested, but in a manner that is arbitrary and need not reflect the relative frequencies with which the various cases are executed. It is possible that the flow of control for the most frequently executed case would have to traverse all the pre-headers.

We adopted the second approach. After loops have been found by the original algorithm, loops that have blocks in common but are not nested are merged into a single loop. This approach does not suffer from the disadvantage outlined in the preceding paragraph. In addition, has the intuitive advantage that in this and similar cases the loops it finds are the ones that the programmer wrote in the first place.

4.2 The Directed Acyclic Graph Representation of a Basic Block

The DAG is one of the fundamental data structures in PCO. It is used to represent the computations in a single basic block. As described by Aho and Ullman[2], the construction process identifies common subexpressions, eliminates redundant assignments, and determines which variables are used in the block. The DAG for a block expresses the values of assigned variables at the end of the block in terms of the values of referenced variables at entry to the block. Each node represents a value: a constant, the value of a variable on entry to the block, or a computation.

The construction of the DAG simulates the execution of the statements in the block. Assignment statements cause identifiers to be attached to nodes that represent the new value. A re-assigned identifier is removed from any previous attachment because that value has been overwritten. At the end of construction of the DAG, the identifiers are attached to nodes that represent the values they must have on exit from the block.

The ultimate use of the DAG is to construct new intermediate code for the basic block. This occurs after various transformations have been done. It is easier to understand some of the issues in DAG construction if the code generation process is kept in mind, so we will describe it briefly. More detail on this process is presented later.

4.2.1 Generating Trees from the DAG The intermediate code output by PCO is a sequence of trees. The trees are constructed bottom-up by processing each node of the DAG in turn. Code for individual DAG nodes is not immediately written to the intermediate file. Output is deferred until a tree representing a C statement is constructed.

If a node has no attached identifiers, it is an "anonymous" subexpression, and no code need be output. If a node has attached identifiers, then the value represented by the node must be assigned to those identifiers. An appropriate assignment statement tree is generated with the identifier on the left hand side and the tree representing the computation of the value on the right hand side. If there is more than one identifier, a "cascaded" assignment is used: `a = b = c + d`. (If this form would cause an incorrect conversion, it is not used.) Note that a subsequent reference to a node with attached identifiers need not re-evaluate the tree. Since the value has been stored in all the attached identifiers, a reference to any of them will obtain the appropriate value.

Anonymous subtrees can be a problem because their evaluation can be deferred until later than in the original program. While this can lead to more efficient code, evaluation order dependencies must be carefully respected. For example, in:

```
t = a + b;
a = c;
c = t + 7;
t = 6;
```

the expression `a + b` is anonymous, and is deferred until the third statement. PCO ensures that the original value of `a` is preserved until it is needed.

4.2.2 Representing Pointer Operations Choosing an appropriate representation of pointer operations in the DAG can be difficult. The design decisions we made here affected many other parts of PCO.

The assignment of a value to a variable is normally represented in the DAG by removing the variable from its current node and attaching it to the node that represents its new value. This simulates the replacement of one value by another.

If the target of the assignment is a dereferenced pointer (a "pointer store"), the situation is more complex. If the current value of the pointer is not known, any one of a number of variables could have values changed by the pointer store. This set of variables is called the "risk set".

Computation of the risk set is difficult, and PCO takes a very conservative approach. Currently, the risk set is deemed to include anything to which a pointer might point: global variables, local variables that have had their addresses taken, and parameters, if the address of any parameter was taken. Under some circumstances it would be possible to determine a more precise risk set. For instance, PCO could identify some indirect stores that modify an element of an array, structure or union. Once this has been done, other variables need not be included in the risk set. This would increase the number of loop invariants PCO can find, since it currently assumes that all indirect fetches are not loop invariant if there is an indirect store anywhere in the loop.

A pointer store is represented in the DAG with an explicit assignment operator, rather than somehow allowing a pointer expression to be on an attached identifier list. A pointer store could not normally be removed from an attached list since, in general, it is not possible to determine if the value has been overwritten. Eliminating the possibility of including arbitrary expressions in attachment lists also simplifies the data structures.

A pointer store in a block creates evaluation order dependencies which must be reflected in the DAG. First, any future reference to an identifier in the risk set must create a new leaf, which represents a new fetch of the value. This means that the same identifier can appear as a leaf several times in the same DAG. In addition, any stores to variables in the risk set must not be deferred past the pointer store.

Similar but simpler considerations arise for pointer fetches (references via a pointer). All stores to variables at risk must be done before the pointer fetch to ensure that the correct value is obtained.

Function calls are similar to pointer operations. The order and number of calls must be preserved. Since the function may do arbitrary pointer operations, all variables in the risk set may be modified or fetched. They must have their correct values at the time of the call, and new fetches must occur after the function returns.

4.2.3 Operators with Side Effects C has several operators that have side effects. These are the pre- and post- increment and decrement operators (`a++`, `++a`, etc.) and the assignment operators (`a *= 0.5`). PCO uses these operators in the generated code whenever it can, even when they did not appear in the input program. Since this transformation is available, we chose to represent expressions using operators with side effects as if they had been written without the special operators. The DAG is marked to ensure that the left-hand-side is evaluated only once.

4.2.4 Conditional Operators C has operators that appear as expressions but effectively cause flow-of-control. PCO represents these conditional operators directly within a DAG. Each conditional operator controls a "conditional zone". These zones are similar to basic blocks, but they can be nested. If one node in the zone is evaluated, all will be. The `&&` and `||` operators have new zones for their right hand side. The `:` operator has a different zone on each side.

Computations within a conditional zone can make use of values already computed. Expressions within a conditional zone might not be evaluated, and PCO ensures that they are not available as common subexpressions outside the conditional.

Side effects within the conditional, such as assignments, might not occur.. If a value computed outside the conditional is assigned within a conditional, the target variable cannot simply be attached to the earlier node. This is prevented by inserting a special node in the DAG that provides a place within the conditional to attach the identifier. The new attachment does not supersede an existing one outside the zone, because the old value might not be overwritten. Subsequent references outside the conditional cause another leaf to be created, because the value must be fetched again.

In order to represent perfectly the semantics of the `?:` operator, the state of the DAG should be the same at the start of the left-hand and right-hand sides of the colon. This allows maximal availability of common subexpressions, but would require that the state of the DAG be saved before the left-hand side is constructed and restored afterward. PCO does not do this, but the effect is minor: expressions computed prior to the evaluation of the `?:` operator that are invalidated by side effects on the left-hand-side of the `:` are not available on the right-hand-side. Correct code is still generated and the slight loss of potential optimizations is insignificant.

Sequences of `&&` or `||` operators often appear in C code. PCO recognises cascaded sequences of conditional operators and preserves the availability of expressions until the entire sequence is complete.

4.2.5 Carriers It can happen that a node with attached identifiers also has a non-zero in-degree, indicating the the value it represents is needed later in the block. It is usually the case that it is better to obtain the value by referencing the variable, rather than re-computing the expression. A variable that can be referenced to get the value represented by a node is called a "carrier" for the node.

Not all attached identifiers are suitable as carriers. If the identifier is re-assigned before the last use of the node to which it is attached, it is not suitable. Global variables are unsuitable if a procedure call or pointer store occurs after assignment to them but before the last reference. During the DAG construction process, information is recorded about when the values of a variable attached to an identifier becomes invalid (e.g. as the result of a pointer store). PCO uses this information to choose a carrier that remains valid until the last reference to the node. If no

suitable carrier exists, a temporary is created.

4.2.6 The Swap Problem: Delayed Stores The very simple code sequence

```
t = a;
a = b;
b = t;
```

turns out to have surprising implications for the DAG representation.

The DAG for this code sequence is:

Node 1: Label: a Attached: b, t
Node 2: Label: b Attached: a

If the simple code generation scheme outlined above is used, the (incorrect) code generated from this DAG is:

```
t = b = a;
a = b;
```

The problem arises any time an identifier is referenced and later assigned a value that was computed prior to the reference. Changing the order of evaluation of the nodes may not help. In the example above, the code would be wrong even if generated in the other order.

In this example, the store into `b` must not occur until the reference to it has occurred, implying that the value to be stored into `b` must be available then. The assignment to `b` is called a "delayed store". With each attached identifier, PCO records a pointer to the most recently constructed leaf reference to that identifier. It is now possible to determine if an assignment would overwrite a value needed later. If it will, the DAG is modified so that during code generation the initial assignment will not be made. The assignment code will actually be generated after the reference to the variable.

4.2.7 The Activity Count Algorithm It is necessary to know when in the evaluation of a basic block there are no more remaining references to a node in the DAG, and to know when an identifier's value changes. This information is determined by the "activity count" algorithm. Aho and Ullman[2] suggest including explicit dependency edges in the DAG. Since dependency information changes as the DAG is transformed, we decided to compute it dynamically whenever it is needed. We have sometimes thought that explicit dependencies would be useful, but the question remains unsettled.

For each node, the algorithm maintains an "activity count", which is the number of outstanding references to the node. Each node is visited in the same order as code generation. When a node is visited, its activity count is decremented to reflect the fact that it has been evaluated. When a node will no longer generate a reference to its children, their activity counts are recursively decremented.

When a node's activity count becomes zero, the node is inactive. Resources associated with the node can be reclaimed and any stores waiting for the node to become inactive can occur.

4.3 More on Generating Trees

The reconstruction of trees from the DAG is essentially as described above, but there is additional special handling for type conversions, delayed stores, and side effects in conditionals.

During code generation, the DAG nodes are processed in the same order that they were created. It would be possible to process them in any order that respected the ordering constraints. In the absence of compelling reasons to prefer one order to another, we chose to avoid the cost of sorting the DAGs.

As each node in the DAG is processed, the appropriate tree is constructed and attached to the node. When a node with attached identifiers or in-degree zero is encountered, the tree attached

to the node is appended to the output intermediate code. If a node is referenced, either the appropriate subtree or a reference to its carrier is generated. Explicit type conversions in the DAG that can be done implicitly (e.g., with an assignment) are removed.

An assignment for a delayed store is generated when no outstanding references to the target identifier exist, which is detected when the "activity count" for the node representing the last reference becomes zero. When this happens, the node being processed will cause code to be output and the delayed store will be output next.

Since conditional operators have expressions as children, side effects that are under the control of these operators must be embedded as expressions within these trees. The side effects within a conditional zone are collected on a stack because these operators can be nested. When the tree for the conditional operator is constructed, the accumulated trees are popped off the stack and are included in the conditional operator's tree. Comma operators are used to enforce the appropriate order of evaluation.

5. Transformations

We now discuss the transformations that PCO makes.

5.1 Unused Variables

For each identifier, PCO determines all places where it appears in a DAG as a leaf. This information is used in a simple scheme to delete unnecessary stores. PCO traverses the DAG of each basic block and examines each attached identifier. If the identifier never appears as a leaf in any DAG, and if it cannot be referenced elsewhere, the store can usually be deleted. The only exception is the use of a temporary to preserve volatile data over a function call or a pointer store.

5.2 Constant Folding

Algorithm 14.1 in Aho and Ullman[2] provides the basis for PCO's constant folding. PCO visits each basic block of the program in turn. Each occurrence of an identifier as a leaf is examined. If the value of the identifier can be determined, and if that value is a constant, then the identifier is replaced by the constant. Each operation is also examined. If its operands are constant, the operator is replaced by the result of the operation.

With the logical operators && and ||, we are even more aggressive. If the left-hand-side of an && operator is 0 (false), or the left-hand-side of an || operator is non-zero (true) then the operator is replaced with 0 (resp. 1) and the right subtree of the operator is erased. (It is never executed.)

This process is iterated until no further changes occur.

5.3 Loop Invariants

Aho and Ullman[2] give two algorithms for moving invariants out of loops. The first algorithm is conservative, moving operations only if they are certain to be executed at least once. The second algorithm does not impose this restriction, and as a consequence may introduce errors into the optimized program. Since the introduction of errors was unacceptable to us, we implemented the first algorithm. We soon found that computations were almost never moved out of loops. Most loops written by C programmers are built using `for()` and `while()` constructs. The termination test is made before the loop is entered, and PCC uses the same code for the termination test after each trip around the loop. Thus, the basic block containing the termination test is the only block that is certain to be executed.

We developed two schemes to allow more code to be moved out of loops. The first of these was suggested by a recent modification to PCC. The compiler can be configured to generate code for a loop test twice — once for the initial test and once for subsequent tests. When this is done, the loop starts after the initial test, and is guaranteed to be executed at least once.

This scheme must be used with some caution. The gains made by detecting and removing loop invariants when the termination test is duplicated must be weighed against the extra code size resulting from duplication. We modified the compiler to duplicate the test only if the expression being tested was fairly simple. The decision is made in a machine specific routine in the compiler, and may be tuned for a given implementation.

Our second scheme was a compromise between the two algorithms of Aho and Ullman[2]. We now allow certain operations to be moved outside a loop even though they might not be executed. Only operations that are immune to error and that have no side effects may be moved. For example, fetches through a pointer are not moved because they might be guarded by tests for a null pointer.

This scheme risks increasing program running time if the body of the loop is never executed. On the other hand, it can decrease running time considerably if the loop is executed many times. As examples of these two extremes, consider the following two code fragments:

```
while((c=getchar()) != EOF) {...
```

and

```
while((c=getchar()) != '\n') {...
```

The two code fragments are nearly identical. However, the first loop probably runs many times, while the second loop might be used to discard unwanted characters on a line, and may never run. PCO might make the second case slower because we assume that loops are traversed more than once.

5.4 Assignment Operators

The assignment operators in C often allow effective use of special hardware instructions. Programmers make heavy use of these operators. PCO makes a pass over the DAG for each basic block, looking for operations that could be done more efficiently using assignment operators or post increment and decrement operators. These might be expressions originally written with assignment operators but they need not be. PCO can generate these operators in situations that the programmer missed.

PCO examines the DAG for configurations of nodes that represent expressions with semantics that can be obtained from use of assignment operators. Evaluation order dependencies and the possible presence of pointer stores or function calls must be taken into account. If it is safe to use an assignment operator, the DAG is modified to reflect this.

5.5 Common Subexpression Elimination

Common subexpression elimination replaces multiple computations of the same value with a reference to a variable in which the value is stored, when that is more efficient. For each subexpression to be eliminated, a temporary variable is created by PCO, and the value of the expression is stored in the temporary. The temporary becomes the carrier of the value, and future references to the expression become references to the temporary.

Within the DAG, a node with in-degree greater than one is eligible. If the node does not already have a carrier, PCO uses a machine-specific function to determine the cost of saving the value and the cost of evaluating the expression. If it is less expensive to save the value and reference it later, a temporary is created and attached to the node. This causes the appropriate assignment statement and references to be generated.

The relative costs depend not only on the number of references to the expression and the cost of evaluating it, but on what kind of storage is allocated for the temporary variable. For some expressions, it is better to save the result if it can be saved in a register, but better to re-evaluate if the result is saved in memory.

The decision on eliminating common subexpressions is made assuming that the temporary will go into a register. If the temporary is not assigned to a register, and it is cheaper to re-evaluate than to reference a local variable, the temporary will not be used.

5.6 Register Allocation

PCO can allocate registers to temporary variables that it has created, and to programmer-declared local variables and parameters. In many cases the programmer has already used the C language `register` storage class, especially in programs that have been tuned for speed. However, improvements can often be made, particularly if the program is being ported to a machine with more registers than the machine on which it was originally written.

Since the programmer can use the `register` storage class, the question arises: how much attention should PCO pay to this declaration? Since PCO's allocation decisions are based on a static analysis, it is possible for it to make unfortunate choices if the program's dynamic behaviour is different from that assumed by PCO. The programmer's register choices may have been based on knowledge of dynamic behaviour, so we honour the programmer's declaration by default. A flag has recently been added to specify that PCO should ignore the programmer's suggestions.

The register allocation algorithm is adapted from Redelmeier.[5] A brief outline is given here. For each object (local variable, parameter, or temporary), there is a possible benefit to putting that object into a register. The benefit is computed based on the machine specific cost functions, and represents the cost savings of a register operation over one using storage.

The algorithm assumes that the lifetimes of the objects it is to allocate nest or are disjoint. The result of the algorithm is a list of sets of objects and associated benefits. The objects in each set can share storage (they have disjoint lifetimes), and the benefit is the sum of the individual benefits. The list is ordered by decreasing benefit. After the vector is computed, PCO goes down the vector allocating objects to registers until all registers are used.

In computing the allocation vector, two operations are used: an element-wise sum when the associated objects have disjoint lifetimes, and a merge when the lifetimes overlap. The final allocation vector is computed by a bottom-up traversal of the tree of object lifetimes, summing and merging vectors as appropriate until the final vector appears at the root.

Temporaries created for common subexpressions and delayed stores are live over some part of a single basic block. PCO determines which temporaries within a block can share storage. Variables to hold loop invariant values are also created by PCO. These are live in every basic block within the loop and in the loop preheader. Local variables and parameters are declared by the programmer. Some may be eligible for promotion to registers. PCO currently considers these objects to be live in all basic blocks in the function.

Local variables that have had their address taken are ineligible for register allocation, as are all parameters if any one has had its address taken.

The computed benefit of putting locals and parameters is biased to reflect the cost of saving the additional register. Parameters have an additional negative bias to reflect the cost of transferring the parameter from the stack to the register.

This algorithm was chosen as a reasonable compromise between optimal allocation and efficient implementation. The assumption that lifetimes of objects are nested allows the algorithm to make local decisions without backtracking. Additional live/dead analysis on function variables will allow more precise co-allocation. For machines on which small stack offsets are more efficient than larger ones, PCO can easily be extended to re-order the stack frame according to the information collected here.

6. *A Fortran 77 Optimizer*

The UNIX Fortran 77 compiler, f77, uses an intermediate code representation that is very similar to PCC. Pass 2 of f77 is almost identical to pass 2 of PCC (they are derived from the same source). Since PCO works with this intermediate representation, only a few changes were required to implement an optimizing f77 compiler. Use of C's assignment operators by PCO provides significant benefits to f77 programs, as does its use of registers.

Some features of Fortran 77 needed special treatment: assigned goto, functions with multiple entry points, and some other minor changes were made. Changes to f77 pass 1 were required to put a symbol table in the intermediate file. The optimizing f77 compiler is now working for the original target architecture, and plans are being made to port it to the VAX.

7. *Results*

In this section, we present the results of running PCO on some benchmark programs: the sieve benchmark,[6] the "puzzle" benchmark, and a multiplication of two 50 by 50 integer matrices. The benchmarks were compiled and run with and without PCO. Figure 1 shows the improvement achieved by PCO on our target machine expressed relative to the results obtained without PCO. A peephole optimizer developed by HCR was used in all cases.

	Improvement on Original Target Machine	
Benchmark	Size	Time
Sieve	50.4%	69.2%
Puzzle	48.1%	69.7%
Matrix	17.4%	53.8%

Figure 1

Figure 2 shows results obtained on HCR's VAX 11/750. The standard VAX peephole optimizer was used in all cases. The VAX compiler was based on PCC from System V Release 2. The machine specific cost function in the VAX version of PCO is still tuned for the original target machine, rather than a VAX. For each benchmark, the first line gives program text sizes in bytes, and the second line gives program execution times in seconds. We have observed that highly tuned C programs may not be improved as dramatically.

		VAX	
Benchmark		Original	With PCO
Sieve	Size	208	148 (28.8%)
	Time	4.2	2.7 (35.7%)
Puzzle	Size	1888	1464 (22.5%)
	Time	22.7	7.8 (65.6%)
Matrix	Size	524	484 (7.6%)
	Time	9.1	3.6 (60.4%)

Figure 2

8. *Future Development*

The first phase of the PCO project, described in this paper, has been completed. Work is underway to tune the existing VAX version, and to port PCO to other architectures. Since PCO is largely machine independent, this process is quick. We are also making additional improvements to PCO. We are improving the handling of arrays to make our information about pointer dereferences more precise. We are extending live/dead analysis to programmer-defined variables to improve register use, and help eliminate useless code. We are also planning to implement global common subexpression elimination.

9. *Conclusions*

We have described an intermediate optimizer for C that transforms the intermediate code of PCC. PCC was generally easy to work with. Our biggest problems were caused by the discarding of important type transfer information. The machine dependent code generated by pass 1 was tedious to remove, but did not pose a serious problem. Basing the optimizer on PCC permitted the development of an optimizing f77 compiler for very little additional work.

The optimizing compiler can significantly improve the performance of C programs, while still maintaining correct semantics. The automatic use of registers for variables can have a very substantial effect on program performance.

PCO allows the existing investment in PCC-based C and f77 compilers to be preserved, while improving the performance of user programs.

10. *Acknowledgements*

We would like to acknowledge the work done on this project by other members of the team: Hugh Redelmeier, Tracy Tims and James Watt. We would also like to thank Mike Tilson and K. W. Lee for their advice.

References

1. **Patterson, D. A.** (1985). "Reduced Instruction Set Computers," *Communications of the ACM* **28**(1), pp. 8-21.

2. **Aho, A. V. and Ullman, J. D.** (1977). *Principles of Compiler Design,* Reading, Mass.: Addison Wesley.

3. **Johnson, S. C.** (1979). "A Tour Through the Portable C Compiler," *Unix Programmer's Manual Volume 2.*

4. **Leffler, S. J.** (1980). "A Detailed Tour Through the /6 Portable C Compiler," Technical Report, Department of Computer Engineering, Case Western Reserve University.

5. **Redelmeier, D. H.** (1976). "A Code Generator For Sue.8," M. Sc. dissertation, Department of Computer Science, University of Toronto.

6. **Galbreath, J. and Galbreath, G.** (1983). "Eratosthenes Revisited: Once More Through the Sieve," *Byte* **8**(1), p. 284.

7. **Muchnick, S. S. and Jones, N. D.** (1981). *Program Flow Analysis: Theory and Applications,* Englewood Cliffs, N. J.: Prentice-Hall.

A Portable Reference Optimizer for the System V Loader

Tracy Tims

Human Computing Resources Corporation
xxihnp4!hcr!tracy

ABSTRACT

Some machine architectures have classes of instructions or operand formats which permit references to memory to be encoded in different ways, where the length of the encoding depends on the address of the referenced object. Most UNIX systems have assemblers which will make near-optimal decisions about addresses which are fully resolved at assembly time. The assembler can make a decision about a reference to an address in the same section (**.text**, **.bss**, or **.data**) of a given file. For external references, or references to a different section, the assembler must write the most general form of the address as it cannot predict how the loader may decide to fix up the unresolved reference. This imposes a speed and space penalty. The general reference format may be significantly slower than more optimal ones, and shortening them to the optimal form may be desirable, especially in programs which are highly modular (many functions) and iterative. This optimization can only take place during the linking of the final program, when all addresses are finally resolved. This paper describes the implementation of an optimizer module for the UNIX* System V loader program, *ld(1)*. The optimizer is implemented in a portable fashion. It shortens the encoding of external references during link editing. It requires little or no assembler modification.

1. Introduction

In many computer architectures, references of a given type to a given address can be encoded in the machine language in more than one way. The use of these encodings is determined by the address of the object being referenced. Usually there is a "general form" of encoding which can be used for all addresses, and there are less general forms of encoding which may be used if the object is in certain areas of memory (absolute addressing) or if it is close enough to the reference (pc relative addressing). An individual reference may be an instruction stream rather than a single instruction.

It is not a trivial problem to decide which encoding should be used. Typical UNIX assemblers make encoding decisions for references to addresses they can completely resolve. For addresses the assembler cannot resolve, the most general form of the reference is written. This implies that references to C functions, and references to objects in the **.data** and **.bss** sections of memory, will always be written in the general form. Some UNIX assemblers can be given flags to force the writing of shorter formats, but this is not a general solution.

We have implemented an optimizer for the UNIX System V loader which optimizes the encoding of external references at load time. This work was done as part of our larger optimizing C compiler project.

---

* UNIX is a trademark of AT&T Bell Laboratories

The original target machine for this work penalizes the general form of an external reference highly, thus making it attractive to optimize the external reference format in the loader. All external addresses of this machine are absolute addresses, and optimization decisions are made on the basis of the absolute address of the object being referenced.

The problem of shortening reference formats is complicated by the obvious fact that any given address is dependent on all preceding shortenings, and any given distance between a reference and its target is dependent on the shortenings of references between them. Algorithms for dealing with this are described in the literature and will not be dealt with in detail here. We use a fairly simple algorithm.

1.1 Relocation

Before any reference shortening can take place, the loader must have the final addresses of all objects, so that it can make meaningful decisions about the format required to store a given address. In the System V loader, the final addresses of references are calculated during the final, output building pass.

Each optimizable reference is a relocatable value. (A relocatable value is almost always an address or offset field of some sort. It may be an instruction stream containing these fields.) For each relocatable value there is an entry in the relocation data table associated with the section which contains the relocatable value. The relocation datum describes the position of the value in the section, its format, and the symbol that it must be relocated with respect to.

In the output pass the value is relocated by applying the relocation rules for the particular format of the value. Generally, the amount of change in the value of the symbol is added to the original relocatable value. The original value is an offset from the value of the symbol. If the reference is to a global symbol, this offset will be zero. (The original value of a global symbol that was not defined in the file that contained the reference is zero. In such a case the effect is to paste the final value of the global symbol into the reference value field.) If the reference is to an non-global object in a given section (for instance an initializer of an auto variable) then the symbol will be the one whose value is the start of that section (in System V these are named **.text, .data** and **.bss**), and the offset will be the offset of the initializer in the section. The offset should be thought of as a separate parameter to the relocation process which just happens to be stored in the field that is the relocatable value (in a hithero unprocessed object file.)

2. Overview

A number of constraints had to be satisfied by the design, as the optimizer was implemented after the formats of the **a.out** file and the assembler conventions had been chosen. These constraints have helped to produce a more portable optimizer. The a.out files are normal COFF (Common Object File Format) files, perhaps with new relocation formats.

The UNIX System V loader is made up of 14,000 lines of source code. Rather than loading in the fashion of the older UNIX loaders, by synchronizing the first and second passes and using dead reckoning to generate addresses; the System V loader builds a large network of linked lists which contain a complete specification in absolute form of the final a.out. This makes it very flexible, and highly complicated[1]. A decision was made to avoid modifying this Brobdingnagian piece of code more than necessary, in order to avoid having to predict and debug the bizarre side effects that would result. The optimizer added almost 2000 lines more to the loader source.

The optimizing code is designed as an "internal post processor". Viewing the optimizing problem as a post processor implies that it is loosely coupled with the loader and therefore highly portable. Placing the optimizer within the loader gives it access to the data structures it needs to make its decisions without giving it large amounts of code for symbol table handling, etc.

The loader is essentially allowed to complete its work and to produce all the data for a regular, non-optimized load. While it does this, the optimizer produces a "address map" which is a

transform from the unoptimized a.out addresses to the optimized a.out addresses. The final action of the loader is to write the a.out file and it is here that the optimizer translates the loader's unoptimized actions into optimized ones. The address map is used at this stage to translate all addresses. This task is complicated by the presence of two addresses spaces: the "file address space" of the a.out file, and the memory address space of the program.

This implementation leaves almost all of the original loader code undisturbed. The most significant changes are the "hooks" for the building of the address maps, and the a.out writing code.

As well as shortening addresses, the loader will also remap the **.data** section to place the greatest amount of data in the lowest addresses. It will only do this if there is an advantage to be gained. This is done to increase the number of data references that can be shortened.

It is possible to write a reference optimizing algorithm which produces an optimal result, given certain constraints on its input[2]. Such an algorithm was not used in order to avoid increasing the complexity of the assembler, and to allow backward compatibility with old object files. An optimal algorithm requires that references be first written as short as possible, and then expanded until the program is correct. This optimizer takes references written in the most general format and it shortens them when such a shortening will be correct. There can exist references which can be shortened which the loader will not be able to detect. We live with this.

2.1 Address Resolution

All addresses must be calculated before the optimizer can make any shortening decisions or reorder the data space. Addresses of all global variables and sections are calculated well before the output pass, but the final values of references are only calculated in the output pass. An extra pass was added to calculate the final reference values ahead of time. This is described below.

3. Design

The optimizing module consists of the text and data space mapping code, the code which builds and optimizes those maps, object file preloading code, and the modifications to the output writing code to use the address mapping.

3.1 The Memory Maps

There are two separate maps used to transform the unoptimized address space, one which describes the **.text** section and the other which describes the **.data** section. Each is implemented by a different set of routines. The data space map is built after final addresses have been chosen for symbols. The text space map is built just after that.

3.1.1 The Data Space Map

The data space map consists of a sorted vector of triples which describe the movement of blocks of memory. Each triple contains an *old address*, a *length*, and a *new address*. A triple denotes that the block of memory of *length* bytes starting at *old address* has been mapped to *new address*.

The triples are sorted according to *old address* and the mapping operation consists of a binary search over the vector to find the appropriate triple, followed by a simple address calculation. The last triple used is cached for performance in sequential lookup situations (this happens when **.data** sections are being written).

The map is constructed in two phases. In the first phase, the triples are collected and placed on a linked list. No address mappings can be performed during this phase. In the second phase, the actual map vector is built from the linked list and sorted. The linked list is freed. Addresses can be mapped after the second phase.

3.1.2 Data Space Map Initialization

The **.data** section is divided into data elements at addresses given by the global data symbols in the symbol table. Each of these elements is atomic and independent from all other elements.

Each element is assigned a score. The current scoring mechanism assigns the highest score to the smallest elements, and the lowest score to the largest element. Starting with the element with the highest score, and preceeding to elements with lower scores, elements are placed in the next best location in the optimized **.data** section by entering a triple onto the data space map. The triple describes the movement of the element from the old location to the new (optimized) location.

There are some introspective UNIX programs which know too much about the internal organization of their **.data** sections. This optimization procedure will confound them. There is a flag to disable data space reordering.

3.1.3 The Text Space Map

The text space map consists of a vector of descriptors. Each descriptor describes one optimizable reference. The descriptors are ordered on the addresses of the references themselves. Each descriptor contains a field giving the original address of the reference itself, the original value of the reference (the place that it referenced), its current format (or length), and the cumulative shortening of the text up to that reference.

A text address is mapped by searching the vector of descriptors for the descriptor of the reference immediately preceding it in the text. The cumulative shortening of the text to that point is subtracted from the old address to give the new address.

3.1.4 Building the Text Space Map

In order to initialize the text space map, a pre-relocation pass must be performed on all the object files that will be loaded. This pass occurs just after final addresses have been chosen for object file sections and for global variables. Its purpose is to determine the final address of each optimizable reference, something that is normally done only during the final output pass.

The relocation data for each object file is searched. For each relocation datum that represents an optimizable reference an entry is made in the text map vector. The initial cumulative shortening is 0 bytes, and the format is the most general format. When a reference is relocated, the actual original reference value in the machine language text must also be read in order to obtain the *offset* mentioned above in the short description of the relocation process.

This procedure results in a complete description of every locus in the text that could possibly change size.

3.1.5 Assembler Support Required

Optimizable references must have a distinct relocation format. It is used to distinguish them from other, more pedestrian references. In our implementation, we have an older format for the general global reference which is unoptimizable, and a new format which marks a global reference as optimizable. The two reference formats themselves are identical. This gives us very useful backward compatibility with old object files.

3.1.6 Optimizing the Text Space Map

Once the text map has been created the optimization procedure proceeds by examining each reference descriptor in the map starting with the one lowest in memory and proceeding upwards. The unoptimized original reference value is mapped using the current state of the text and data maps. If the new value can be represented in a shorter format the new format is recorded in the descriptor for that reference. A count of the total number of bytes of shortening on the current pass is kept: after each examination of a descriptor the current value is added to the cumulative shortening value of the descriptor. Note that when this is done it will invalidate all cumulative shortening values in higher descriptors. This will cause some non-optimal decisions to be made,

which will be fixed up in the next pass. Multiple passes are mqde over the map until no more changes in the length of the **.text** section occur.

After each pass the starting address of the **.data** section is adjusted downwards (due to the text shortening) depending on alignment constraints. This new starting address of the data is taken into account on the next pass.

3.2 The Output Pass

During the output pass the loader places all sections from input object files into the corresponding output section in the output file.

3.2.1 The Map Interface Routines

Two sets of map interface routines exist. One set maps file addresses into optimized file addresses, and the other set maps memory addresses into optimize

d memory addresses. The routines decide which map (text space or data space) to use depending on the original address. Thus one function can be called to obtain an address translation without any knowledge of the whereabouts of the original address.

3.2.2 Writing Values

There are several kinds of literal values in the output file which need to be mapped. The header of the output file contains file offsets to the sections of the file, and to the symbol table. Each relocation datum describes a field which makes a reference to a memory address. For each of these cases a call to the correct mapping routine (file address or memory address) is made to translate the value before the value is written.

There are many other instances where it is necessary to write an address value into the file. Examples include *sdb(1)* support symbol table entries. In all of these cases, producing the correct optimized value is simply a matter of determining whether or not it is a file space0address or a memory space address, and using the correct mapping routine before the write.

3.2.3 Writing Section Data

Several changes were made to the code that actually writes the section data. In an unoptimized load the section data for a given section in a given object file can be written in a contiguous chunk. This is not true in the case of an optimized load, because there may be shortenings (in the case of the **.text** segment) or wholesale reordering (in the case of the **.data** segment).

The code for writing output sections was changed to support the writing of each individual byte of section data at an arbitrary location specified with each byte. In actuality, it is unnecessary to go this far. The **.text** sections are written contiguously, and where an optimizable refer

ence is encountered the appropriate (possibly shorter) form is written, according to the decision in the text map. The **.data** section writing code performs a map translation on the address of each byte to be written to the output file. Since the mapped addresses of the bytes of each data element are sequential, the caching mechanism on the data mapping routine obviates the need for an expensive mapping lookup on every byte. A seek on the output file is only done when the file address being written to deviates from sequential addresses.

3.3 Atomicity of Sections

The Common Object File Format assumes that a section of an object file (of which the **.text**, **.data** and **.bss** sections are examples) are atomic entities. That is, the internal relations of the section will never change. The optimization destroys this property.

This property is usually taken advantage of in text sections by writing references to the same section (if the machine language supports it) in a PC relative format. Normally, no relocation datum needs to be written for these references because the relationship between the referencer and the referencee never changes. The optimizing loader will change these relationships by shortening reference formats between a reference and its target.

If the assembler writes such references, it must also write relocation data describing them, so they will be correctly relocated after optimization. Relocation formats may have to be added to support this. The output writing code must relocate these formats via the maps.

This is one reason why it is important to have a non-optimizable relocation format for the general form of a reference, if backward compatibilty is desired. The older object files will not contain these "local" relocation data, and the long form references in them must not be optimized. Defining a new relocation format for optimizable references confers this property.

4. Portability

Since the optimizer module attempts to interact with the loader logic as little as possible (and succeeds) it is not only possible to port it between versions of the System V loader for different machines; it is also possible to port it between different loaders. The memory mapping implementation provides a complete description of every byte of the optimized a.out file, eliminating special cases.

5. Other Possible Applications

On some computers that have architectures that make them poorly suited to C, it may be possible to gain a significant advantage with the optimizing loader. For example, consider a computer which had an inefficient method of addressing global data. It might be possible on such a machine to dedicate a base register to point at the base of the **.data** section, and to use it with a shorter addressing mode to yield a small amount of fast global data address space. The optimizing module of the loader would be used to put the most frequently accessed data into that space, and to optimize the references to it. This could result in significant performance improvements.

With greater modifications to the compiler and the assembler, it might be possible for a program flow optimizer to tell the loader about global data that occured in loops that were highly nested, or global data that the optimizer expected to be accessed frequently. The loader would then preferentially place such data into the fast data space. Such a technique would be possible with literal values and initializers that were frequently used.

6. Summary

The optimization module integrates in a fairly transparent way with the System V loader. With no user knowledge of the system, it will improve the quality of the executable program. We have seen typical improvements in running speed from 4 to 10 percent, depending on the amount of other optimizations performed by other programs. It is clear that for some architectures, global reference optimization can make measurable improvements, producting optimal or near optimal reference lengths.

7. Acknowledgements

I would like to acknowledge the contributions made to this program and its design by Tom Kelly and Allen McIntosh.

References

1. **Bell Laboratories** (1983). "Common Link Editor Reference Manual," .
2. **Szymanski, Thomas G.** (1978). "Assembling Code for Machines with Span-Dependent Instructions," *Communications of the ACM* **21**(4), p. 300.

Improving the Performance of Scientific Applications on a Supermicro Using A Custom Floating Point Processor and An Optimizing Compiler

Curt Gridley

Massachusetts Computer Corp.
Westford, MA 01886

ABSTRACT

A custom floating point processor and a globally optimizing Fortran 77 compiler combine to give Masscomp's 68010-based MC-500 UNIX† system floating point performance exceeding that of many traditional minicomputers for typical scientific applications. The optimizing compiler improves the performance of scientific applications that use the floating point processor by up to 2 to 1 over non-optimized versions. In addition, the Fortran compiler is the only known version of the standard Unix F77 compiler officially validated to be in full compliance with Fortran 77 standards. Architectural descriptions of the floating point processor and the optimizing compiler are given, along with performance statistics and some observations concerning the significance of various design features.

1. Introduction

Many new supermicro computers designed around VLSI processors exceed minicomputers and mainframe computers in their price-performance ratio. They also rival minicomputers in absolute non-floating point computational power. However, the limited availability of high performance floating point coprocessors has restricted the ability of supermicros to compete with larger computers in terms of absolute floating point performance.

Since most scientific applications contain significant amounts of floating point calculations, it is important that supermicros competing in the scientific computing market be able to offer floating point performance that equals or exceeds that of minicomputers.

This paper describes the architecture of a custom floating point processor and an optimizing Fortran compiler that combine to offer floating point performance exceeding that of many traditional minicomputers. We first describe Masscomp's FP-501 floating point processor, which maintains a simple architecture while using the latest in bit-slice technology.

In the later sections, we discuss Masscomp's optimizing Fortran 77 compiler, which is a descendent of the standard Unix F77Fel78a compiler. We briefly describe the optimizations done by the compiler as originally ported from the 4.2 BSD F77 compiler. We then detail some of the significant changes made for a subsequent release of the compiler. Finally, we make observations about the importance of various FP-501 architectural features and compiler optimizations on floating point performance. Concluding remarks suggest directions for future development and improvement.

† UNIX is a Trademark of Bell Laboratories.

2. Hardware Overview

2.1 MC-500 System Architecture

The Masscomp FP-501 floating point processor (FPP) is an optional peripheral processor available with Masscomp's MC-500 Unix-based family of computers. The host processor is a Motorola 68010 operating at 10 Mhz with a 4 kb cache. The FPP uses an AMD2901 bit-slice micro-engine and a 16x16 bit multiplier on a single board to achieve high performance at low cost.

The MC-500 family is designed around a triple-bus architecture to maximize system throughput. The FPP is connected to the CPU via a high speed memory interconnect (MI) bus, which has a peak throughput of 8 Mbytes per second. The memory subsystem and a peripheral array processorHew85a are also connected to the MI bus. Storage peripherals, such as disk and tape controllers, reside on a separate enhanced Multibus. Data acquisition peripherals connect to a separate enhanced IEEE STD+ bus.

In a dual processor (DP) system, two CPUs, each with their own floating point processor and local memory on separate MI busses, may exist in a single system. In certain environments, the DP configuration can result in over twice the floating point throughput of a single processor system. This can be explained by the fact that the second CPU does not handle device interrupts and, therefore, can provide a higher percentage of its processing power to user processes. The DP system is described in an earlier paperFin85a .

2.2 FP-501 Architecture

The FP-501 floating point processor is a memory mapped device that occupies three 4 kb pages of memory at the top of each process's virtual address space. One page of virtual address space, called the FPP Instruction Page, is accessed by the host 68010 to initiate FPP instructions. A second page, called the FPP Register Page, is mapped to a set of 32 64-bit virtual registers. These registers occupy the first 256 bytes of the Register Page. The rest of the page is ignored by the FPP. The FPP Status Page, which contains condition code and status registers, occupies the third page of virtual address space.

Though each user process can access only a single set of 32 virtual registers, there are really 32 sets of physical registers, each set consisting of 32 64-bit registers, for a total of 1024 physical registers. The association of virtual registers with physical registers will be explained in the next section.

The host CPU interacts with the FPP over the MI bus by reading and writing memory locations within the FPP's three pages of virtual address space. These interactions involve initiating instructions as well as loading and storing FPP registers, including FPP exception and status registers. The only external operations the FPP initiates directly involve signaling exceptions by posting interrupts on the MI bus and invoking CPU-FPP synchronization by limiting MI bus access.

2.2.1 FP-501 Register Set

Each FPP register consists of 8 bytes of 100 nanosecond memory that can be read or written directly by the host processor as well as by the FPP micro-engine. The upper four bytes of each register can hold a single precision value or the upper half of a double precision value. The lower four bytes can hold a 16 or 32-bit integer or the lower half of a double precision value. This allows each register to hold a single precision value and an integer value simultaneously.

Although there are 32 separate physical register sets on the FPP, each user process is allocated only a single set of 32 physical registers. Since 8 of the 32 physical register sets are reserved for use by the micro-engine as scratch registers, 24 sets of physical registers are available for allocation to user processes. This allows up to 24 processes to simultaneously use the FPP without

any need for saving and restoring FPP registers during context switches.

A user process is allocated a unique *physical* register set when it first attempts to access the FPP. However, a user process always references the same *virtual* register set. The FPP micro-engine interprets all references by a process to the virtual register set as being references to the unique physical register set that is currently *active*. The FPP Selection Register specifies which physical register set is active at any given time. It is the responsibility of the operating system to insure that the FPP Selection Register points to the correct physical register set associated with the currently running process. During a context switch, the operating system changes the FPP Selection Register to point to the physical register set associated with the next process.

2.2.2 FP-501 Instruction Set

Besides the FP-501's large register set, its architecture features a simple and uniform three operand instruction set that operates only on register operands. The instruction set includes basic arithmetic operations - using single precision, double precision, and integer values - value conversion instructions, single precision transcendental instructions, and miscellaneous move and status instructions. The table below lists the FP-501 instruction set.

FP-501 Instruction Set

Instruction Class	Operation	Instructions
Arithmetic	Add	*addf, addd*
	Subtract	*subf, subd*
	Multiply	*mull, mulf, muld*
	Divide	*divl, divf, divd*
	Mod	*divl*
	PolyStep	*polyf, polyd*
	MoveAbsolute	*mabsf, mabsd*
	MoveNegate	*mnegf, mnegd*
Conversion	TruncateToInt	*intfl, intdl*
	RoundToInt	*cvtfl, cvtdl, cvtfw*
	Float	*cvtwf, cvtlf, cvtdf*
	Double	*cvtld, cvtfd*
Transcendental	Sine	*sinf*
	Cosine	*cosf*
	Square Root	*sqrtf*
	Natural Log	*lnf*
	Common Log	*logf*
	Exp	*expf*
	Arctan/Arctan2	*atanf, atan2f*
Miscellaneous	Compare	*cmpf, cmpd*
	MoveReg	*movf, movd*
	Nop	*nofpp*
	ReadExceptions	*rdecs*
	LoadExceptions	*ldecs*
	ReadRound	*rdrnd*
	LoadRound	*ldrnd*

To start an FPP instruction, the host 68010 writes to a particular location in the FPP Instruction Page. Each address in the Instruction Page corresponds to a different FPP instruction. The data word that is written to an instruction address specifies the register operands the

FPP uses when executing that instruction.

For example, the FPP Instruction Page address associated with the single precision add instruction is 0xffe100. The assembler instruction

addf f0,f1,f2

for adding the single precision values in registers f0 and f1 and placing the result in register f2, is translated by the assembler into a 68010 move-word immediate-value to short-absolute-address instruction:

movw #0x8402, 0xffe100.w

The FPP opcode for the *addf* instruction is represented by 7 bits in the short-absolute destination address, *0xffe100.w*. FPP register operands *f0, f1, and f2,* used by the *addf* instruction are encoded using 15 of 16 bits in the immediate-data-word, *#0x8402*. Each register operand is represented using 5 bits in the immediate-data-word. Representing the registers for a 3 operand instruction, therefore, requires 15 bits. In addition to the 7 opcode bits and 15 register operand bits, a single bit in the destination address toggles *explicit* CPU-FPP synchronization. This will be explained in the next section. Thus, a total of 23 bits of FPP opcode, operand, and synchronization information are transferred in a single, fast 68010 *movw* instruction.

2.2.3 CPU-FPP Synchronization

A 10 Mhz 68010 can execute the above *movw* instruction in 16 cycles or 1.6 microseconds. The FPP, executing instructions asynchronously with the host, can also execute simple instructions, such as the single precision add instruction, in 16 cycles. However, many FPP instructions take longer than 16 cycles to execute. Therefore, a synchronization mechanism is provided to keep the host from interrupting a previous FPP instruction. The table below gives an indication of the execution times for FPP instructions.

FP-501 Instruction Times

Instruction	Precision	Time (in microseconds)
Add/Subtract	single	1.6
Multiply	single	1.6-1.7
Add/Subtract	double	2.0-3.0
Multiply	double	4.5-5.0
Sine	single	20-30

A single FPP instruction buffer allows overlap of instruction execution and instruction initiation or register loads and stores. When the instruction buffer fills up, the FPP locks the MI bus and keeps the host from initiating more FPP instructions or accessing the FPP register set. However, the host can continue executing non-FPP instructions in its instruction cache until the FPP instruction buffer empties and the MI bus is released. This form of synchronization is referred to as *implicit* synchronization.

The host can also set a bit in an individual FPP instruction to invoke the synchronization mechanism directly, whether the instruction buffer is full or not. This guarantees that the instruction will be completed before the host is allowed to fetch the instruction results from an FPP register. The compiler attempts to use this *explicit* synchronization mechanism only when absolutely necessary so that maximum overlap occurs for instruction execution and CPU-FPP interaction.

3. Masscomp Fortran 77 Compiler

The Masscomp optimizing Fortran 77 compiler is a descendant of the standard Unix Fortran 77 compiler, originally written by Stu Feldman at Bell Labs and later improved by students at the University of California at Berkeley. We have extensively reworked this compiler over the past 18 months to improve its reliability, functionality, and performance.

As a partial result of this work, in October 1984 the compiler was validated by the Federal Software Testing Center to be in full compliance with the ANSI Fortran 77 standard. This is one of the few Fortran compilers that has been validated on a Unix system and the only validated compiler that we know of that is based on the original Unix F77 compiler.

Around the same time, we ported the optimizer from the 4.2 BSD Fortran compiler, which provided performance improvements of up to 60%. Recently, this optimizer was reworked and additional optimizations were incorporated which resulted in a combined average improvement of approximately 2 to 1 over the non-optimizing compiler. This level of performance has equaled or exceeded the performance of a DEC VAX-11/750† with floating point accelerator and DEC VAX-11 Fortran.

We will first describe the general structure of the Fortran compiler and the optimizations it did before we ported the 4.2 BSD optimizer. We note that the structure of the standard Unix Fortran compiler presents several roadblocks to implementing significant optimizations. We describe several of these problems later in the paper and indicate how we resolved some of them when we ported the optimizer from the 4.2 BSD compiler to our machine.

After describing the 4.2 BSD optimizer, we will discuss other optimizations and code generation improvements we implemented to take better advantage of the FP-501 architecture. Finally, we provide performance statistics and observations about the importance of the various FP-501 architectural features and compiler optimizations.

3.1 General Compiler Architecture

The Masscomp Fortran compiler consists of 6 separate programs. The main startup program for the compiler is the *f77* driver. It parses command line arguments, initializes files, and controls execution of the other compiler components.

The second program is *f77pass1*. It embodies the front and middle ends of the compiler, including the most important optimization phases. It scans and parses Fortran statements into an intermediate language (IL) for semantic checking. If optimization is not enabled, *f77pass1* will then translate the IL expressions representing a Fortran source statement into another intermediate language (C-trees) and write them to a file. If optimization is enabled, the IL is buffered on a procedural basis so that optimization can be done on the entire procedure before the IL is translated into C-trees and written to a file. C-trees closely resemble C expression trees and use expression operators modeled after C operators.

The third component of the compiler is the code generator, called *fort*. It reads C-trees from the file created by *f77pass1* and generates 68010 assembler code to an assembler file. *Fort* is very similar to the code generator for the standard Unix C compiler, generally referred to as *pcc* - the *portable C compiler*.Joh81a In fact both *fort* and *pcc* are built from the same source files using conditional compilation.

If optimization is enabled, the assembler file produced by the code generator is processed by the peephole optimizer *c2*. *C2* eliminates redundant loads and stores of registers and jumps to jumps, among other things. The output of *c2* is then assembled by the system's 68010 assembler, *as*. The object file output by the assembler can be combined with other object files and library routines, using the linker *ld*, to produce an executable image.

---

†DEC and VAX are trademarks of Digital Equipment Corporation.

3.2 Previous Optimizations

Before porting the 4.2 BSD optimizer to the MC-500, we used a version of *f77pass1* from System III Unix. This version was nearly the same as the original Unix Fortran front end. It did perform several minor optimizations beyond a straight-forward translation of Fortran statements into IL expressions. These optimizations included:

- allocating integer DO loop indices to registers,
- local constant folding for simple operations,
- generating some intrinsic functions inline rather than generating calls to subroutines,
- replacing integer multiplies by constant powers of 2 with shifts,
- reducing exponentiation by small integer constants to multiplies and adds,
- applying arithmetic identities to simplify expressions,
- short circuiting boolean expressions, i.e. minimizing the number of tests executed before a branch is taken or control falls through.

These optimizations provided only minor performance improvements. No attempt was made to optimize array references or to do extensive register allocation. For a language such as Fortran, which relies heavily on arrays for most applications, array references within loops can account for a significant amount of the execution time of a typical application. In such instances, optimizations such as common subexpression elimination, removal of invariant code from loops, and global register allocation are essential for achieving fast execution times.

3.3 Weaknesses in Unix F77 Compiler

The standard Unix Fortran 77 compiler has several structural weaknesses that make it difficult to provide global optimizations. These weaknesses include the following:

- It lacks a common symbol table throughout the separate compiler components.

When doing global optimizations, it is useful to have a single source for accumulating information about each variable in a procedure. Much of the information required is contained in the symbol table in *f77pass1*. However, this information is not passed on to subsequent components of the compiler. With this structure, a global optimizer needs to be part of *f77pass1* to access this information.

- It has an intermediate language oriented more towards C.
- It compiles programs one statement at a time.

The standard Unix Fortran compiler overcomes certain restrictions imposed by the IL by generating a combination of assembler code and IL expressions at the same time. When the IL lacks a certain functionality, *f77pass1* generates assembler code directly instead. Procedure entry points are handled in this way. To do significant optimization, the code for an entire procedure needs to be represented in a reasonable internal form and buffered so that multiple passes can be made over it.

- The execution order is not clearly defined in the IL.

Most optimizing compilers use a form of intermediate language called *tuples*. This is a linear representation of a program's statements and expressions so that the execution order of expressions is explicit. *Tuples* are generally easier to analyze and manipulate for most optimizations than are expression trees.

Ideally, there should be a common intermediate language and a common symbol table format accessed not only by the different parts of the Fortran compiler, but also by compilers for other languages such as C. This would allow for a common optimizer as well as a common code

generator and common compiler utilities. The standard Unix compilers have some of this commonality, but they suffer from a weak IL and the lack of a common symbol table format.

3.4 In Search of the Almighty Optimizer

To achieve significant improvements in code quality in the shortest possible time, we decided to port the optimizing front end from the 4.2 BSD Fortran compiler. This version of *f77pass1,* which runs on VAXs, contains a code buffering scheme and several optimization phases that were added by students at the University of California at Berkeley. Most of the code for the 4.2 BSD version of *f77pass1* was the same as in our existing version so we were familiar with its structure. The 4.2 BSD version did contain some VAX hardware dependencies, however, especially in the register allocation phase of the optimizer. These dependencies had to be removed or replaced and several miscellaneous bugs had to be fixed to port the front end.

The initial port, done in the spring of 1984 for release 2.1B of the compiler, took roughly one person-month to complete. However, this initial port did not enable the optimization phases. Getting the optimizer to work took 1-2 additional person-months. This was done in the fall of 1984 for release 3.0 of the compiler.

Nearly all the additional 1-2 months was spent porting and debugging the register allocation phase of the optimizer. This phase originally consisted of over 1800 lines of dense, complicated, and uncommented C code. By the time we were done with release 3.1 in February 1985, it consisted of over 3300 lines of C code, including one or two comments.

Many of the problems we encountered were not directly related to the port but represented fundamental bugs in the register and temporary allocation phases of the optimizer. The register allocation phase continues to be the most troublesome part of the optimizer to maintain.

3.5 4.2 BSD Compiler Optimizations.

We will briefly outline the major optimizations contained in the original optimizer as ported from the 4.2 BSD compiler. While the optimizer in the 4.2 BSD compiler does not overcome all the impediments to significant optimization mentioned earlier that exist in the standard Unix F77 compiler, it manages to resolve some of them by extending the IL in the front end and buffering this extended IL on a procedural basis. This provides the foundation necessary to do some significant optimizations.

3.5.1 Code Buffering Scheme.

When optimization is not enabled, the front end of the compiler translates Fortran source into C-trees and writes them to an output file one statement at a time. To do any significant optimizations, a mechanism for buffering the IL expressions was required. A simple scheme is used whereby each ordinary IL expression is allocated a *slot* in a buffer. The buffer is simply a doubly linked list of slots. Each slot is marked with a type specifier that indicates the type of information contained in the slot. Slot types include GOTO, IF_FALSE, EXPRESSION, CALL, LABEL, ASSIGN, DO_HEAD, and DO_END.

The addition of this second level of type information to the old expression-based IL allows the new extended IL to represent the entire procedure in a consistent way while still retaining the old IL at a lower level. By simply extending the old IL, a minimum amount of change was required to the rest of the compiler. Later optimization phases operate directly on the extended IL in the buffer. After optimization, the buffer slots are translated into C-trees and assembler code and written to the output file, the same as when optimization is not enabled.

3.5.2 Removing Loop Invariant Code.

Removal of invariant code from loops is the first major optimization phase applied during optimization of a routine. Expressions involving only variables whose values do not change within a loop are moved by the optimizer to just before the beginning of the loop. This process continues from inner loops to outer loops so that it is possible for an expression that is invariant within several nested loops to bubble out to just before the outermost loop.

In removing invariant code from loops, the optimizer assumes that the routine being optimized does not contain nonstandard control flow. In particular, it assumes that a routine does not contain a GOTO statement *outside* a DO loop that has a label destination *inside* a loop. In other words, it assumes control flow does not enter a DO loop except through the DO statement at the top of the loop. With this assumption, the optimizer can determine if any variable occurring within a loop is loop invariant by examining only the statements within the loop. However, Fortran 66 allows *extended* DO loops, whereby statements outside the lexical scope of a loop can be executed while iterating through the loop. The 4.2 BSD optimizer may attempt to remove code from such loops that is not really invariant during iterations of the loop.

3.5.3 Extended Temporary Allocation.

Normally, temporary variables allocated on the stack to hold intermediate results are freed after their last lexical use. The variable is then available to be re-allocated. However, removal of invariant code from loops can result in instances where the last lexical use of a temporary is within a loop, but the temporary can not be freed until the end of the loop, since it may be referenced again on subsequent loop iterations. Hence, a slightly more general temporary allocation scheme is required that can retain temporaries through their last lexical use to the bottom of a loop.

The compiler handles this problem by associating a special buffer slot with each temporary indicating when the temporary can be freed. If the optimizer removes a temporary variable from a loop it will also move the corresponding FREE_TMP slot to just after the bottom of the loop. This delays any possible re-allocation of the temporary until after the end of the loop.

3.5.4 CSE Elimination.

The optimizer does common subexpression elimination after it has removed invariant code from loops and has divided the IL for a procedure into contiguous segments called *basic blocks*. A *basic block* is a maximal single-entry, single-exit section of code. For basic blocks, the compiler can accurately determine the data flow for variables within each block. The algorithm the optimizer uses for CSE elimination is patterned after the value numbering scheme given in *Principles of Compiler Design*, by Aho and UllmanAho78a

3.5.5 Copy Propagation.

After common subexpression elimination the optimizer does copy propagation. Removal of invariant code from loops and common subexpression elimination can create copy statements of the form *temp1 = temp2,* where *temp1* and *temp2* are temporary variables created by the optimizer. If *temp2* is assigned a value only once, and the only assignment to *temp1* is the statement *temp1 = temp2,* the optimizer can replace all references to *temp1* with references to *temp2*. *Temp1* can then be eliminated altogether. Besides removing an assignment statement, copy propagation eliminates an extra temporary variable, thus reducing memory and register requirements for a procedure.

3.5.6 Register Allocation.

The final phase of the optimizer is the register allocation phase. The register allocator works from inner loops to outer loops assigning registers to the most used variables. Thus, variables in the innermost loops are more likely to be allocated registers than variables only referenced in outer loops.

This phase of the optimizer was difficult to port from the VAX to the 68010 because of the latter's non-uniform register set. The 4.2 BSD version relied on each register functioning equally well as an address register or as a data register. The 68010 differs in this respect from the VAX by having different registers for each function. The problem was solved by running the register allocator three separate times: once for data registers, once for address registers, and once for FP-501 floating point registers. Implementing and debugging this code was a painfully slow process.

3.5.7 Base Pointer to Local Variables

A mechanism was added to the storage allocator in *f77pass1* so that the local variables of a procedure would be grouped together and allocated contiguous memory locations. When a compiled procedure is executed, the base address of the local variables for that procedure is assigned to a register and the local variables are accessed using an offset from the base register. This mode of addressing variables is slightly faster and requires less space than the absolute addressing mode previously used.

3.6 New Optimizations for FP-501

Once the 4.2 BSD optimizer was up and running, we had a chance to consider further optimizations. We discovered several ways we could improve performance significantly for programs that made heavy use of the FP-501 floating point processor. These changes required roughly 2 person-months to implement and debug. They were added to version 3.1 of the compiler, which was released in February 1985. Most of the changes are described below.

3.6.1 Better Use of Temporary Registers

Due to the large number of registers available on the FPP, we decided very early in the development of compiler support for the FPP to reserve the upper six registers, f26-f31, for use only by important math library routines. These library routines, written in assembler language, do not call any other routines and so can use the upper six registers as local scratch registers without having to save and restore them. In release 3.0 of the Fortran compiler, these registers were not allocated at all by the compiler.

When we established the register convention for library routines, we recognized that the same convention could be applied to compiled routines that do not call other routines: a Fortran routine that does not contain any subroutine or function calls can also use the upper six FPP registers without needing to save or restore them. This convention divides the FPP registers into two classes: registers that are saved by the calling routine; and those that are saved by the called routine. The upper six FPP registers are classified as being saved and restored by the calling routine. Fortran routines that do not call any other routines can use these registers without any overhead of having to save and restore them.

This can have a significant impact on programs that contain small low-level routines called many times from within loops. The impact would be even more noticeable if the optimizer were applied to a more modern structured language where such small routines are more prevalent.

3.6.2 Extended Register Allocation

The register allocation phase was originally applied only to nested DO loops. The register allocation phase was not applied to routines that did not contain any DO loops. Such routines can often benefit from allocation of the upper six registers as described in the last section, since the time spent saving and restoring registers can be a significant part of the time spent executing the routine. Therefore, in the 3.1 compiler, when compiling a procedure that does not contain any DO loops, a phantom DO loop is put around the procedure so that the register allocation phase will be run once over the entire procedure.

3.6.3 Allocating Arguments to Registers

The ANSI Fortran 77 standard states that a single variable may not be bound to two separate arguments of a procedure call if both arguments are changed by the call. Otherwise, there would be confusion over which change really applied to the variable after return from the procedure call. This allows arguments to a procedure to be assigned to registers without changing the behavior of a valid Fortran 77 program, since we can be sure that an assignment to one argument variable within the procedure cannot affect any other argument variable. In other words, there cannot be any aliasing between arguments in a standard conforming Fortran 77 program.

The 4.2 BSD optimizer did not take advantage of this fact and so we changed the compiler for the 3.1 release to potentially allocate procedure arguments to registers. However, since it is difficult to catch infractions of the above rule, assigning arguments to registers can potentially change the behavior of an (invalid) Fortran 77 program without notifying the user (1) that their program is invalid, and (2) that it will not behave correctly when compiled using the optimizer. So far we have not encountered any problems with this change, however.

3.6.4 Removing Loads and Stores

When the optimizer has allocated a variable to a register for the duration of a loop, it must load its value from memory into the register before the start of the loop and store its value from the register into memory at the end of the loop. For temporary variables and variables that have not been assigned a value before they are assigned to a register, loading the value of the variable into the register at the start of the loop is not necessary. The same is true for storing a register variable at the end of a loop if the variable is not used after the loop before being assigned a new value. In the 3.1 release of the compiler, we changed the optimizer to be more careful before creating such unnecessary loads and stores.

3.6.5 Reducing Explicit Synchronization

By default, the compiler always uses the explicit synchronization mechanism for the FPP. This mechanism was described in the first part of this paper. Always using explicit synchronization guarantees that the result of an FPP instruction will be available before it is accessed by the CPU. However, this eliminates the possibility of overlapping FPP instruction initiation with instruction execution.

In the 3.1 compiler, we improved on the simple criterion of always using explicit synchronization by removing the explicit synchronization from the first of every two contiguous FPP instructions. By not explicitly specifying synchronization for the first instruction we are able to overlap the loading of the second instruction with the execution of the first instruction.

Due to an architectural feature of the FPP, we do not have to worry about a later FPP instruction interrupting an earlier FPP instruction when they are contiguous instructions. The FPP will implicitly invoke the synchronization mechanism if the first instruction is still executing when the second instruction is loaded into the instruction buffer. This inhibits any subsequent instructions from accessing the register set to obtain an instruction result until the first instruction finishes and the second instruction can be removed from the instruction buffer.

3.6.6 Using Three Operand Instructions

In the 3.0 compiler, the code generator did not take advantage of three operand FPP instructions. For instance, if *A, B,* and *C* are all bound to FPP registers, the statement

$$A = B + C$$

requires only a single FPP instruction. It was not until the release 3.1 of the compiler that we were able to take advantage of this feature, however.

In addition, the special three operand PolyStep instruction, which implements

$$C = C + A * B,$$

where *A, B,* and *C* are all in FPP registers, was first used by the compiler in the 3.1 release. Now this instruction is generated frequently, especially in programs that calculate the inner product of two arrays or do matrix row-reduction operations.

3.6.7 Generating Intrinsics Inline

The FPP implements commonly used single precision transcendental instructions, such as *sine* and *cosine,* in micro-code. For the 3.0 compiler, these instructions were accessed using function calls to routines in the math library. For the 3.1 release of the compiler, we created a special intrinsic operator in the IL to specify transcendental functions that could be generated inline. Marking these functions with a special IL operator not only saves the significant overhead of function calls by generating the instructions inline, but it also increases the ability of the optimizer to optimize expressions involving COMMON block variables.

For example, consider the following Fortran program:

```
      Program Main
      Common /FOO/ xvar
100   x = xvar + y
      call Subr
200   z = xvar + y
      end

      Subroutine Subr
      Common /FOO/ xvar
300   xvar = 2
      return
      end
```

The subexpression *xvar+y* occurring in statements 100 and 200 cannot be treated as a common subexpression since the value of *xvar* is changed by the call to *Subr.*

Whenever the optimizer encounters an ordinary function or subroutine call, it must assume that the routine called will change the value of all COMMON block variables. This rule must be applied unless the optimizer knows *a priori* that the routine called will not change the value of any variables in COMMON. This is the case when transcendental functions are called. The new IL operator for transcendental functions added for release 3.1 allows the optimizer to mark transcendental functions as special and, hence, to more effectively optimize expressions involving COMMON block variables in the presence of calls to transcendental functions.

3.6.8 Faster Instruction Initiation

Before the 3.1 release of the compiler, FPP instructions had to be initiated using a 32-bit *movl* instruction rather than a 16-bit *movw* instruction. The extra 16 bits were needed by the operating system to support emulation of FPP instructions on older 68000-based systems that did not have an FPP. These extra bits were ignored by the FPP. The longer *movl* instruction takes 24 cycles to execute rather than the 16 cycles required for the *movw* instruction. For the 3.1 release, minor changes were made to the FP-501, operating system, and assembler, that allowed both forms of FPP instruction initiation to be used. We have observed performance improvements of up to 8% simply as a result of using the short form of instruction initiation.

3.6.9 Reducing *Calloc(3C)* Overhead

The Fortran compiler calls the dynamic memory management routines *calloc(3C)* and *free(3C)* in a disorganized manner in *f77pass1*. This did not prove to be a problem until the 4.2 BSD optimizer was added for release 3.0 of the compiler. The optimizer makes heavy use of these memory management routines. The disorganized mix of calls to these routines combined with the increased use of dynamic storage by the optimizer resulted in a tremendous growth in the number of page faults during the compilation of some routines.

In one case, on a system with only one megabyte of memory, compiling a large routine with many nested DO loops resulted in over 150,000 page faults. This increased the compilation time for this routine by a factor of over 20. We have since centralized the use of *calloc(3C)* and *free(3C)* and created separate free lists for each structure allocated. This has eliminated the problem.

3.7 Performance Results

The table below lists the execution times for several benchmarks compiled with the 3.0 and 3.1 compilers. The 2.1B compiler performs the same as the non-optimized version of the 3.0 and 3.1 compilers. The 3.0 release of the compiler was the first version that incorporated substantial optimizations from the 4.2 BSD compiler. These optimizations were described in Section 3.5. The 3.1 release contained significant improvements over the 3.0 compiler due to some additional optimizations, as explained in Section 3.6.

FP-501 and F77 Compiler Benchmark Results

	Whetstones (KWhets)	Linpak (Kflops)	Mflop (Kflops)	FFT-1024 (Milliseconds)
V2.1B (NoFPP/Opt)	110	12	15	???
V2.1B (FPP/Opt)	525	53	62	360
V3.0 (FPP/Opt)	575	81	85	290
V3.1 (FPP/Opt)	920	110	112	206
V3.1 (FPP/Opt)		170 (BLAS)		

The table above includes the following benchmark programs:

- Whetstones - A widely used synthetic benchmark that was designed to represent average instruction mixes for scientific applications. This benchmark contains several small subroutines that are called many times within loops. This program benefitted from all the optimizations described in Section 3.6. The relative performance of the Whetstones is measured in KWhets per second.

- Linpak - Demonstrates the performance of some core subroutines contained in a well known linear algebra package. Most of the time in this benchmark is spent in a routine that performs matrix row reductions. This program was significantly improved by putting arguments that were the base addresses of arrays in registers and using the PolyStep instruction. The entry marked *BLAS* is for a version of the program that has had its core routines carefully handcoded in assembler language. This provides us a measure of the effectiveness of using the current optimizing Fortran compiler relative to handcoding in assembler language. The major difference between the code for the two versions is that the hand coded version uses auto-increment addressing and careful instruction scheduling for maximum overlap of instruction execution and register loads and stores. Performance is measured in thousands of floating point instructions per second, or kiloflops.

- Mflop - A version of a benchmark from Los Alamos Scientific Labs that contains the inner loops from several scientific applications.

- FFT-1024 - Measures the time required to do a 1024 point single precision FFT. This is a frequent operation in most signal processing applications.

3.8 Observations and Conclusions

The 3.0 compiler project was Masscomp's first attempt at global optimization in any of its compilers. This effort consisted of porting the optimizer from the 4.2 BSD Fortran 77 compiler to our existing Fortran compiler. This port resulted in significant performance boosts for many applications. Encouraged by the results, we tried to improve, within the framework of the recently ported optimizer, particular instances of poor code quality. This effort resulted in the successful 3.1 compiler release. However, with the significant performance improvements already achieved, we feel we have realized the limits of what this sort of an optimizer is capable of achieving reliably. So where do we go from here?

The existing optimizer is severely hampered during optimization by its lack of an explicit control flow graph. There are also many powerful optimizations which it does not attempt, such as induction variable elimination, dead code elimination, constant propagation, procedure wide register allocation, and various interprocedural optimizations. Others have attempted to alter the standard Unix C and Fortran compilers to do such optimizationsLas83a, Ell82a with varying degrees of success. Given the many inherent weaknesses of the older Unix compilers, such as fixed internal table sizes, slow compilation, no common symbol table format, and an IL which is not conducive to global optimization, we feel a new compiler effort is warranted rather than another attempt to stretch the limits of the existing compilers.

We are currently developing a new family of compilers that will share a common symbol table format, common front end utilities, a common IL, a common global optimizer, a common code generator, and a common assembler. Each compiler will consist of a single program rather than several separate programs. This improves the commonality of the components of the compiler and improves compilation speeds by eliminating the communication overhead between separate components. Compilers for different languages will be supported by developing separate front ends for each language. Initially, this will include compilers for C and Fortran. Eventually it should include compilers for other languages as well. This will allow us to get maximum leverage for our investment in the common parts of the compilers. In particular, we are going to be paying special attention to generating the highest quality code possible. As we implement some of the optimizations mentioned earlier using global data flow analysis techniques, other optimizations such as code scheduling to maximize instruction overlap for the FPP will become

increasingly important.

The FP-501 has continued to be a solid and consistent performer. One of the original design goals was for a processor that could achieve 600 KWhets. As the previous table shows, it has easily surpassed this goal. Its simple architecture and more than ample register set make it an easy processor for which to generate code. The recent increases in the performance of chips that can be used to construct a floating point processor such as the FP-501 makes it appear unlikely that architectural changes will be required to achieve large increases in overall floating point performance. The FP-501 continues to compare favorably against the floating point coprocessors that are currently becoming available for microprocessors.

4. Acknowledgements

The people involved in the initial design and development of the FP-501 and related compiler and operating system support included Dave Cane, Curt Gridley, Paul Guilbault, Terry Hayes, Dominic Li, Don Monroe, Mike Ross, and John Sundman. They all deserve praise for their ingenuity, their tireless effort, and their amazing capacity for staying awake during the regular Monday morning meeting. I also want to thank the reviewers of this paper. They accepted a thankless job on very short notice and provided valuable criticisms of an earlier draft.

Aho78a. AHO, A. AND ULLMAN, J., *Principles of Compiler Design*, Addison-Wesley, Reading, Massachusetts (March 1978).

Ell82a. ELLIS, M. A., "Global Data Flow Analysis in a Fortran 77 Compiler," in *unpublished master's report*, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California (September 1982).

Fel78a. FELDMAN, STUART I. AND WEINBERGER, PETER J., "A Portable FORTRAN 77 Compiler," in *Unix Programmers Manual*, The Bell Telephone Laboratories (November 12, 1978).

Fin85a. FINGER, E. J., KRUEGER, M., AND NUGENT, A., "A Multiple CPU Version of the UNIX Kernel," *1985 Winter USENIX Conference Proceedings* (January 1985).

Hew85a. HEWSON, D., CULLEN, G., AND NUGENT, A., "Integral Array Processing in a Multiprocessor UNIX Environment," *1985 Summer USENIX Conference Proceedings* (June 1985).

Joh81a. JOHNSON, S. C., "A Tour Through the Portable C Compiler," *Bell Laboratories Memorandum for File* (January 1981).

Las83a. LASCHKEWITSCH, M. AND POWERS, G., "Compiling Globally Optimal Code," *Systems & Software* (August 1983).

The USENIX Association

USENIX, the UNIX and Advanced Computing Systems professional and technical organization, is a not-for-profit membership association made up of systems researchers and developers, systems administrators, programmers, support staff, application developers and educators.

USENIX is dedicated to:

* fostering innovation and communicating research and technological developments,
* sharing ideas and experience, relevant to UNIX, UNIX-related and advanced computing systems
* providing a forum for the exercise of critical thought and airing of technical issues.

Founded in 1975, the Association sponsors two annual technical conferences and frequent symposia and workshops addressing special interest topics, such as C++, distributed systems, Mach, systems administration, and security. USENIX publishes proceedings of its meetings, a bi-monthly newsletter *login:*, and a refereed technical quarterly, *Computing Systems*. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts.

There are four classes of membership in the Association: student, individual, institutional, and supporting, differentiated primarily by the fees paid and services provided. The supporting members of the Association are:

Digital Equipment Corporation
Frame Technology, Inc.
Matsushita Graphic Communication Systems, Inc.
mt Xinu
Open Software Foundation
Quality Micro Systems
Rational Corporation
Sun Microsystems, Inc.
Sybase, Inc.
UNIX System Laboratories, Inc.
UUNET Technologies, Inc.

For further information about membership or to order publications, contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710-2565
Telephone: 510/528-8649
Email: office@usenix.org
Fax: 510/548-5738